

UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
BENTAHAR AMINE

IDENTIFICATION DES PRÉOCCUPATIONS TRANSVERSES :
UNE APPROCHE STATIQUE BASÉE SUR UNE ANALYSE DU FLOT DE CONTRÔLE

MAI 2012

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

Résumé

La programmation orientée objet a été largement utilisée dans les milieux industriels. Les nombreuses retombées d'expériences ont révélé les limites sérieuses qu'elle éprouve. Ces limites sont essentiellement dues à la difficulté de représenter les préoccupations transverses, qui recoupent plusieurs parties d'un système. Le code correspondant à ces préoccupations est souvent dupliqué et dispersé dans les programmes. Ceci affecte leur modularité et rend difficile, entre autres, leur compréhension, leur test ainsi que leur maintenance.

Une des solutions envisagées consiste à utiliser la programmation orientée aspect. Elle permet, en effet, de factoriser le code correspondant aux éléments transversaux des systèmes en des unités modulaires séparées appelées *aspects*. La migration d'un système orienté objet vers un système orienté aspect passe d'abord par l'identification et la localisation des préoccupations transverses dans le code objet (*aspect mining*) pour les représenter, par la suite, par des aspects (*aspect refactoring*).

L'objectif principal de ce projet étant d'explorer une approche statique (par analyse du code) basée sur une analyse des chemins de contrôle (algorithmes des méthodes, dépendances en termes d'appels et de contrôle entre méthodes de classes, chemins d'exécution, ...) pour la détection de code pouvant correspondre à des préoccupations transverses.

Il s'agit, en particulier, d'utiliser les principes de deux techniques dynamiques existantes, basées sur les traces d'exécutions (analyse des appels récurrents et analyse formelle de concepts), et les adapter à une analyse statique du code basée sur une analyse des chemins de contrôle. L'idée étant d'extraire, par analyse statique d'un code objet (applications Java), des formes récurrentes dans les chemins de contrôle. Les deux techniques résultantes ont été évaluées empiriquement, à l'aide de plusieurs critères, sur quelques exemples et une étude de cas concrète d'envergure. L'étude de cas considérée a subi plusieurs activités de *refactoring aspect*. L'évaluation empirique a également intégré, pour des fins de comparaison, les résultats fournis par une troisième technique statique (*Fan-in*) connue dans la littérature du domaine.

Abstract

Object-oriented programming has been widely used in industrial fields. The many benefits of experiments revealed its limitations. These limitations are mainly due to the difficulty of representing crosscutting concerns. The code corresponding to these concerns is often duplicated and dispersed in programs. This affects their modularity and makes difficult their comprehension, test and maintenance.

One solution is to use aspect-oriented programming. It allows, indeed, factoring the code corresponding to the transverse elements of the systems in separate modular units called aspects. Migration of an object-oriented system to an aspect-oriented system pass initially by the identification and localization of crosscutting concerns in the object code (aspect mining) to represent them thereafter by aspects (aspect refactoring).

The main objective of this project was to explore a static approach (static analysis of the code) based on an analysis of control paths (algorithms of methods, dependencies in terms of calls and control between methods, execution paths, etc.) for detection of code that may correspond to crosscutting concerns.

This is, in particular, to use the principles of two existing dynamic techniques, based on execution traces (analyzing recurring patterns of execution traces and formal concept analysis of execution traces), and adapting them to a static analysis of the code based on an analysis of control paths. The idea is to extract, by static analysis of an object code (java

applications), recurrent forms in the control paths. The resulting two techniques have been evaluated empirically, using several criteria, on some examples and a case study of practical scale. The considered case study has undergone several activities of aspect refactoring. Empirical evaluation has also included, for comparison purposes, the results provided by a third static technique (*Fan-in*) known in the literature of the domain.

Remerciements

En préambule à ce mémoire, je remercie Dieu « الله » qui m'a donné le courage pour terminer ce travail et la force d'atteindre mon but. Je souhaiterais aussi adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire ainsi qu'à la réussite de ces formidables années universitaires.

Je tiens à remercier sincèrement Monsieur Mourad Badri, qui, en tant que Directeur de recherche, s'est toujours montré à l'écoute et très disponible tout au long de la réalisation de ce mémoire, ainsi que pour l'inspiration, l'aide et le temps qu'il a bien voulu me consacrer et sans qui ce mémoire n'aurait jamais vu le jour. Mes remerciements s'adressent également à Madame Linda Badri : codirectrice, pour sa générosité et la grande patience dont elle a su faire preuve malgré ses charges académiques et professionnelles. Je n'oublie pas mes parents ainsi que ma future femme pour leur contribution, leur soutien et leur patience. Je tiens à exprimer ma reconnaissance envers les jurys qui ont eu la gentillesse de lire et corriger ce travail.

Enfin, j'adresse mes plus sincères remerciements à mes frères, ma sœur, mes proches et amis, qui m'ont toujours soutenu et encouragé au cours de la réalisation de ce mémoire. Merci à tous et à toutes.

Table des matières

Résumé.....	ii
Abstract	iv
Remerciements.....	vi
Table des matières.....	vii
Liste des tableaux.....	xi
Liste des figures	xii
Liste des symboles	xiv
Chapitre 1 - Introduction.....	1
Chapitre 2 - Aspect Mining : techniques et outils.....	4
2.1 Introduction	4
2.2 Aspect Mining	5
2.3 Les différents types de techniques.....	7
2.3.1 Les navigateurs dédiés	7
2.3.2 Identification (semi-) automatique des aspects candidats.....	8
2.4 Approches.....	8

2.4.1	La technique d'analyse des appels récurrents et transverses basée sur les traces d'exécution	8
2.4.2	La technique d'analyse formelle de concepts basée sur les traces d'exécution	9
2.4.3	Traitement du langage naturel	10
2.4.4	Détection des méthodes uniques	11
2.4.5	La technique Fan-in	11
2.5	Classification des techniques.....	13
Chapitre 3 - La technique d'analyse des appels récurrents et transverses basée sur les scénarios.....		
3.1	Introduction	15
3.2	Algorithme de la technique	15
3.2.1	Génération des scénarios.....	16
3.2.2	Classification des relations d'exécution.....	19
3.2.3	Les contraintes des relations d'exécution	22
3.2.4	L'identification des appels récurrents et transverses	24
Chapitre 4 - La technique d'Analyse Formelle de Concepts (FCA) basée sur les scénarios26		
4.1	Introduction	26
4.2	Algorithme de la technique	27
4.2.1	Génération des scénarios.....	27

4.2.2 Contextes Formels	28
4.2.3 Concepts.....	29
4.2.4 Treillis de concepts	29
4.2.5 Identification des preoccupations transverses.....	31
4.3 Application de la technique sur un exemple.....	32
Chapitre 5 - Évaluation empirique	37
5.1 Introduction	37
5.2 Exemple 1 (Design Pattern Observer).....	38
5.2.1 Présentation de l'exemple	38
5.2.2 Évaluation de la technique d'analyse des appels récurrents et transverses basée sur les scénarios.....	41
5.2.3 Évaluation de la technique d'analyse formelle de concepts basée sur les scénarios	47
5.3 Exemple 2 (la classe TangledStack).....	51
5.3.1 Présentation de l'exemple	51
5.3.2 Évaluation de la technique d'analyse des appels récurrents et transverses basée sur les scénarios.....	52
5.3.3 Évaluation de la technique d'analyse formelle de concepts basée sur les scénarios	56
5.4 Exemple 3 (Design pattern chaîne de responsabilité)	60

5.4.1	Présentation de l'exemple	60
5.4.2	Évaluation de la technique d'analyse des appels récurrents et transverses basée sur les scénarios.....	62
5.4.1	Évaluation de la technique d'analyse formelle de concepts basée sur les scénarios	65
5.5	Etude de cas.....	69
5.5.1	Terminologies	70
5.5.2	Définitions.....	71
5.5.3	Application des techniques d'aspects mining	73
5.5.4	Étude statistique	81
5.5.5	Conclusion	84
Chapitre 6 - Conclusion générale.....		86
Références		88
Annexe A – Code source de l'exemple 1 (Design Pattern Observer) du chapitre 5		91
Annexe B – Code source de l'exemple 2 (la classe TangledStack) du chapitre 5		94
Annexe C – Code source de l'exemple 3 (Design pattern chaîne de responsabilité) du chapitre 5.....		96

Liste des tableaux

Tableau 1 Fan-in des méthodes de la figure 3 [8].	12
Tableau 2 La liste des techniques comparées.	13
Tableau 3 Classification des différentes techniques automatiques d'aspect mining.	14
Tableau 4 Contexte formel sur certains animaux.	28
Tableau 5 Contexte formel d'un ensemble d'objets O et d'attributs A [15].	30
Tableau 6 Les deux classes Arbre_Binaire et Nœud.	32
Tableau 7 Relation entre cas d'utilisations et les appels de méthodes [15].	33
Tableau 8 Contexte formel de l'arbre binaire.	34
Tableau 9 Abréviation des noms des méthodes.	48
Tableau 10 Contexte formel de l'exemple 1.	48
Tableau 11 Abréviation de noms des méthodes.	56
Tableau 12 Le contexte formel de l'exemple 2.	56
Tableau 13 Abréviation des noms des méthodes.	65
Tableau 14 Le contexte formel de l'exemple 3.	66
Tableau 15 Résultats de la technique des appels récurrents (précision et rappel) en fonction du seuil.	74
Tableau 16 Résultats de la technique FCA (précision et rappel) en fonction du seuil.	77
Tableau 17 Résultats de la technique Fan-in (précision et rappel) en fonction du seuil.	79
Tableau 18 Résultats des différentes combinaisons (précision et rappel).	81
Tableau 19 Résultats des corrélations de Pearson et Spearman et l'AUC (courbe ROC).	82
Tableau 20 Corrélations et l'AUC de la technique Fan-in avec seulement 23 classes détectées non réellement aspectualisées.	83

Liste des figures

Figure 1 Migration d'un système Orienté Objet vers un système Orienté Aspect [5].	6
Figure 2 Trace d'exécution.	8
Figure 3 Divers appels de la méthode polymorphe m [8].	12
Figure 4 Construction des graphes de contrôle réduit aux appels.	17
Figure 5 Contrôle réduit aux appels de plusieurs méthodes.	18
Figure 6 Chemins d'appels compactés.	18
Figure 7 Chemins d'exécutions possibles.	18
Figure 8 Exemple pour illustrer les relations d'exécution.	20
Figure 9 L'ensemble des concepts du contexte formel présenté dans le tableau 5.	30
Figure 10 Exemple de treillis de concepts [15].	31
Figure 11 Les concepts définis du tableau 8.	34
Figure 12 Treillis de concept de l'arbre binaire [15].	35
Figure 13 Diagramme UML du pattern Observer [20].	39
Figure 14 Code source du modèle observateur.	40
Figure 15 La liste des chemins d'exécutions de l'exemple 1.	41
Figure 16 Les relations d'exécutions de l'exemple 1.	42
Figure 17 Les relations vides de l'exemple 1.	43
Figure 18 Les relations d'exécution uniformes de l'exemple 1.	44
Figure 19 Les relations d'exécutions uniformes et transverses.	45
Figure 20 Les aspects candidats détectés par la technique 1 de l'exemple 1.	46
Figure 21 L'ensemble des scénarios de l'exemple 1.	47

Figure 22 Les différents concepts de l'exemple 1.....	48
Figure 23 La liste finale des concepts de l'exemple 1.....	49
Figure 24 Treillis de concepts de l'exemple 1.....	49
Figure 25 Les aspects candidats détectés par la technique 2 de l'exemple 1.	50
Figure 26 Code de la classe TangledStack [21].	52
Figure 27 L'ensemble des chemins d'exécution de l'exemple 2.	53
Figure 28 Les relations d'exécutions de l'exemple 2.....	54
Figure 29 Les relations d'exécutions transverses de l'exemple 2.	55
Figure 30 La liste des concepts de l'exemple 2.....	57
Figure 31 Le treillis de concepts de l'exemple 2.....	58
Figure 32 Les aspects candidats détectés par la technique 2 de l'exemple 2.	59
Figure 33 Exemple d'implémentation du patron chaîne de responsabilités (la classe <i>ColorImage</i>) [21].	61
Figure 34 La liste des chemins d'exécution de l'exemple 3.	62
Figure 35 Les relations d'exécution de l'exemple 3.	63
Figure 36 Les relations d'exécution uniformes et transverses de l'exemple 3.....	64
Figure 37 La liste des concepts de l'exemple 3.....	66
Figure 38 Le treillis de concepts de l'exemple 3.....	67
Figure 39 Les aspects candidats de l'exemple 3 détectés par la technique 2.	68
Figure 40 Précisions et rappels de la technique des appels récurrents en fonction du seuil.	75
Figure 41 Précisions et rappels de la technique FCA en fonction du seuil.....	77
Figure 42 Précisions et rappels de la technique Fan-in en fonction du seuil.	80
Figure 43 La courbe roc de la technique fan-in avec seulement 23 classes détectées et non réellement aspectualisées.	84

Liste des symboles

AOSD	Aspect-Oriented Software Development
AM	Aspect Mining
FCA	Formal Concept Analysis
NLP	Natural Language Processing
OO	Object-Oriented (Orienté-Objet)
ROC	Receiver Operating Characteristic (caractéristique de fonctionnement du récepteur)
AUC	Area Under Curve (aire sous la courbe)

Chapitre 1 - Introduction

Les techniques de programmation ont beaucoup évolué durant les dernières décennies. La programmation orientée objet a été largement utilisée aussi bien dans les milieux industriels qu'académiques. Le paradigme objet fournit une meilleure concordance avec les problèmes du monde réel.

Malgré tous les avantages et les caractéristiques du paradigme objet, tels que l'encapsulation, le polymorphisme et l'héritage, il souffre de quelques insuffisances. Il ne permet pas de bien saisir toutes les décisions de conception importantes [1]. Il s'agit principalement des exigences non fonctionnelles (sécurité, sûreté, convivialité, concurrence, etc.). Ces exigences posent souvent problème, car il est difficile d'en limiter la portée dans un module bien circonscrit. On se retrouve donc avec des problématiques dont l'implémentation se retrouve un peu partout dans les différents modules fonctionnels du système.

Tout système informatique peut être vu comme étant un ensemble de préoccupations. Chaque préoccupation correspond soit à une tâche métier (exigence fonctionnelle) ou une tâche applicative (exigence non fonctionnelle). Généralement, les exigences fonctionnelles sont réparties dans des modules bien séparés. Par contre, les exigences non fonctionnelles sont difficilement prises en compte et on se contente de les intégrer dans les différents modules fonctionnels. C'est ce qu'on appelle des préoccupations transverses (*crosscutting concerns*) [2]. En conséquence, tout cela rend difficile la réutilisation des classes. Il en

résulte, aussi, que pour faire une modification sur certaines fonctionnalités de l'application, nous devons toucher plusieurs lignes de code dispersées dans différentes classes (*scattering concerns*). Généralement, le code des préoccupations transverses est entremêlé avec le reste du code dans chaque module. Ce problème est connu sous le nom de « *tangling concerns* ».

Pour résoudre ces problèmes, plusieurs chercheurs se sont intéressés à ce qu'on appelle l'*aspect mining*, pour identifier et localiser ces préoccupations transverses dans le code et l'*aspect refactoring*, pour écrire chaque préoccupation transverse dans un module séparé appelé *aspect*. Ceci permet d'améliorer la modularité des applications (avec tous les avantages que cela procure), et la migration des systèmes du paradigme orienté objet vers le paradigme orienté aspect [3]. Ce dernier apporte une solution élégante et simple à ce genre de problèmes. Cette nouvelle approche dans le développement permet d'implémenter chaque problématique indépendamment des autres.

Dans ce mémoire, nous allons nous concentrer uniquement sur l'*Aspect Mining*. Ce domaine est typiquement décrit comme un processus de réingénierie. Les systèmes orientés objet (OO) existants (code source) sont étudiés (analysés) afin de découvrir certaines parties du système qui peuvent être représentées en utilisant des *aspects* [4].

Le domaine de l'*aspect mining* offre des techniques et des outils permettant l'analyse statique ou dynamique du code source des applications afin de localiser les endroits où il y a dispersion ou enchevêtrement. Ces parties constituent, selon les techniques, des parties de code candidates à restructuration.

Parmi les techniques dynamiques, nous avons la technique basée sur une analyse des appels récurrents et celle basée sur une analyse formelle des concepts. Ces techniques sont basées sur l'analyse des traces d'exécution.

Notre objectif consiste à utiliser les principes de ces deux techniques et de les adapter à une analyse statique du code basée sur une analyse des chemins de contrôle. L'idée étant d'extraire, par analyse statique d'un code objet (applications Java), des formes récurrentes dans les chemins de contrôle. Les deux techniques résultantes ont été évaluées empiriquement, à l'aide de plusieurs critères, sur quelques exemples et une étude de cas concrète d'envergure. L'étude de cas considérée a subi plusieurs activités de *refactoring aspect*. L'évaluation empirique a également intégré, pour des fins de comparaison, les résultats fournis par une troisième technique statique (*Fan-in*) connue dans la littérature du domaine.

Le reste de ce mémoire est organisé comme suit : Au chapitre 2, nous présentons un survol des principales techniques (et outils) du domaine de l'*aspect mining*. Ensuite, nous introduirons dans les deux chapitres 3 et 4 qui suivent les deux techniques obtenues suite aux adaptations effectuées. Le chapitre 5 présente une première évaluation sur des exemples simples mais néanmoins concrets, ensuite une étude empirique que nous avons effectuée, basée sur une étude de cas réelle, ainsi que les résultats obtenus. Nous terminerons le mémoire par une conclusion générale.

Chapitre 2 - Aspect Mining : techniques et outils

2.1 Introduction

Le développement orienté aspect fournit de nouveaux concepts permettant l'écriture des préoccupations transverses dans des modules séparés appelés *aspects*. Presque 15 ans après sa conception initiale, cette nouvelle technologie a maintenant quitté le laboratoire de recherche où elle a été conçue et commence à être adoptée par l'industrie. Cela pose de nouvelles problématiques de recherche intéressantes. En particulier, le besoin de techniques et d'outils permettant la migration des systèmes existants (orientés objet en particulier) à une solution orientée aspect. L'étude et le développement de ces techniques est l'objectif des domaines de recherche « *Aspect Mining* » et « *Aspect Refactoring* » [5] :

- Aspect Mining : détection et localisation des préoccupations transverses qui pourraient potentiellement être transformées en aspects. Nous pourrions les appeler aussi aspects candidats.
- Aspect Refactoring : séparation des aspects candidats (code correspondant à des préoccupations transverses) dans des aspects.

Dans ce travail, nous nous intéressons uniquement à l'*aspect mining*.

2.2 Aspect Mining

L'adoption industrielle du paradigme OO dans les années quatre-vingt dix a conduit à un besoin de migration des systèmes existants (à l'époque) vers une solution OO et une augmentation subséquente de la recherche sur la *rétro-ingénierie* du logiciel, la *réingénierie*, la restructuration et le *refactoring*. La même chose se passe actuellement pour le paradigme orienté aspect. Pour migrer des systèmes existants, en particulier des systèmes orientés objet, vers des systèmes orientés aspect, il existe deux types de solutions. Soit nous utilisons des navigateurs dédiés qui peuvent aider les développeurs à naviguer manuellement dans le code source. Ces navigateurs localisent certains endroits dans le système qui peuvent être soit des fonctionnalités dispersées ou entremêlées. Une autre solution consiste à offrir des techniques et des outils ayant pour objectif d'automatiser l'identification des aspects candidats.

Les raisons de migrer un système existant vers une solution orientée aspect sont multiples. Dans un système OO, les mécanismes de *modularisation* disponibles ne permettent pas d'écrire certaines fonctionnalités dans des modules bien séparés. En conséquence, ces fonctionnalités sont dispersées et/ou entremêlées dans le code source. Cela rend la compréhension, la maintenance et l'évolution de ces systèmes plus difficiles. En utilisant la technologie orientée aspect [6], les préoccupations transverses peuvent être clairement séparées dans des aspects, ce qui rend le système plus facile à maintenir et à faire évoluer.

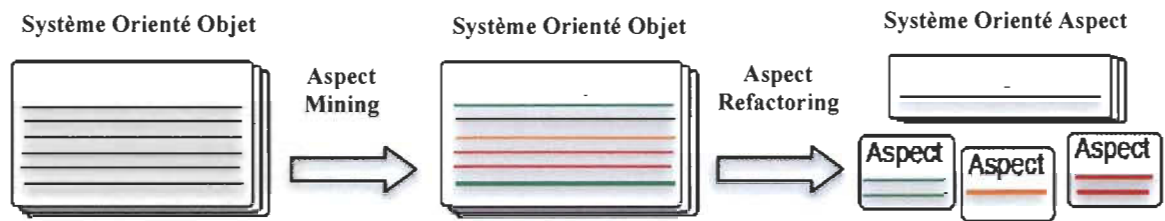


Figure 1 Migration d'un système Orienté Objet vers un système Orienté Aspect [5].

Comme mentionné dans la figure 1 ci-dessus, le processus de migration d'un système OO vers un système OA passe par deux étapes : l'identification des aspects candidats et la *refactorisation* de ces candidats en *aspects*. Dans cette recherche, nous nous intéressons aux techniques et outils qui mettent l'accent sur l'identification des aspects candidats possibles. Ceci n'est pas une tâche facile, en raison de la taille et de la complexité des systèmes actuels et l'absence de la documentation explicite sur les préoccupations transverses présentes dans ces systèmes [5].

Comme indiqué dans l'introduction, nous définissons l'*Aspect Mining* en tant qu'activité qui consiste à découvrir, dans le code source d'un logiciel donné, les préoccupations transverses qui pourraient potentiellement être transformées en *aspect*. La plupart des articles publiés dans ce domaine semblent d'accord sur cette définition (par exemple, [7] et [8]).

Bien que ce domaine est encore à ses débuts, nous pensons qu'une étude comme celle-ci est pertinente et nécessaire car :

- De nombreux chercheurs travaillent sur l'*aspect mining*.
- Plusieurs techniques sont proposées, la plupart sont prématurées.

- Certains chercheurs ont identifié la nécessité de la combinaison entre les différentes techniques [9].

2.3 Les différents types de techniques

Comme mentionné ci-dessus, le processus de migration des systèmes existants vers des systèmes orientés aspects passe par deux étapes : l'identification des aspects candidats et la réécriture de certains de ces candidats dans des aspects. Dans cette section, nous étudions les deux principaux types d'approches d'aspect mining :

2.3.1 Les navigateurs dédiés

Les navigateurs dédiés sont conçus pour aider le développeur à naviguer manuellement dans le code source d'un système donné pour explorer les préoccupations transverses. Typiquement, un utilisateur débute par un point de départ dans le code source. Il utilise le navigateur pour explorer davantage une préoccupation. Pour ce faire, le navigateur propose d'autres points qui pourraient être liés à la préoccupation transverse. D'autres navigateurs fournissent un langage de requête afin de localiser manuellement la préoccupation [5]. Parmi ces approches, nous citons « *Concern Graphs* » proposée par Martin P. Robillard et Gail C. Murphy [10]. Ils ont développé un outil d'analyse et d'exploration de fonctions (*FEAT*) pour représenter les préoccupations extraites d'un système Java sous forme d'un graphe permettant à un développeur de manipuler ce genre de représentation. Il existe d'autres approches comme *Intensional Views* [11], *Aspect Browser* [12], et *JQuery* [13], etc.

2.3.2 Identification (semi-) automatique des aspects candidats

Il existe un certain nombre de techniques ayant pour objectif d'automatiser le processus d'identification des aspects candidats. Ces techniques sont basées soit sur le code source (analyse statique) ou sur des données générées après l'exécution du système (analyse dynamique). Dans ce travail, nous allons nous concentrer uniquement sur ce type de techniques (reliées à nos travaux).

2.4 Approches

Nous allons présenter, dans ce qui suit, les principales approches d'*aspect mining* automatisées qui ont été proposées ces dernières années.

2.4.1 La technique d'analyse des appels récurrents et transverses basée sur les traces d'exécution

Les deux chercheurs Breu et Krinke ont proposé une technique d'*aspect mining* nommée *DynAMit* [14]. Cette technique analyse les traces d'exécution, indiquant le comportement d'exécution d'un système, en cherchant les modes d'exécutions récurrentes (répétitives). Pour ce faire, ils ont introduit la notion de relations d'exécution entre les appels de méthodes. Prenons comme exemple la trace d'exécution suivante :

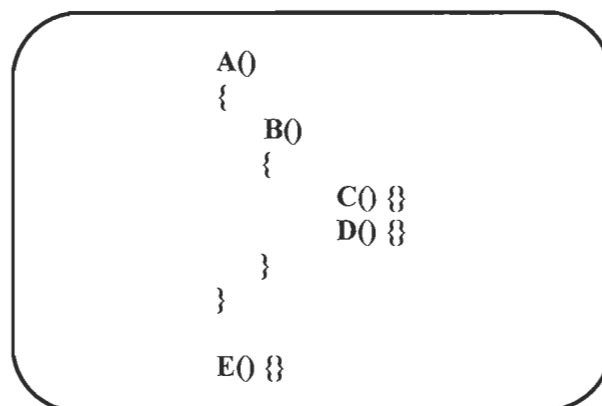


Figure 2 Trace d'exécution.

Breu et Krink distinguent 4 types de relations d'exécution :

- ✓ Outside-before (La méthode A est appelée avant la méthode E)
- ✓ Outside-after (La méthode E est appelée après la méthode A)
- ✓ Inside-first (C est la première méthode appelée dans la méthode B)
- ✓ Inside-last (D est la dernière méthode appelée dans la méthode B).

Grace à ces relations d'exécution, leur algorithme identifie les aspects candidats en se basant sur les appels récurrents. Si une relation d'exécution se produit plus d'une fois, et se répète d'une façon uniforme (par exemple, chaque invocation de la méthode A est suivie par l'invocation de la méthode E), elle est considérée comme un aspect candidat. Bien sûr, pour s'assurer qu'une telle relation d'exécution représente un aspect candidat, il y a une condition supplémentaire reliée au fait que cette relation récurrente devrait apparaître dans différents contextes d'appel.

Dans ce mémoire, nous allons proposer une adaptation de cette technique. L'approche résultante est basée sur une analyse statique (analyse des chemins de contrôle).

2.4.2 La technique d'analyse formelle de concepts basée sur les traces d'exécution

Tonella et Ceccato [15] ont développé *Dynamo*, une technique basée sur l'analyse formelle de concepts (FCA) des traces d'exécution pour identifier des aspects candidats possibles. Le principe de cette technique est de définir un ensemble de concepts. Chaque concept C est un ensemble d'objets O (cas d'utilisations) et un ensemble d'attributs A (méthodes) décrivant ces objets. Nous disons qu'un concept $C1 = (O1, A1)$ spécialise un autre concept $C2 = (O2, A2)$, si l'ensemble d'objets O1 du concept C1 est inclus dans l'ensemble d'objets O2 du concept C2 et inversement (l'ensemble des attributs A2 du

concept C2 est inclus dans l'ensemble des attributs A1 du concept C1). Les concepts sont représentés par un treillis de concepts. Un treillis de concepts L représente chaque concept par un nœud et chaque couple de concepts (C1, C2) par un arc où C1 spécialise C2. À l'aide des concepts et des treillis de concepts, nous détectons les aspects candidats si les deux contraintes suivantes sont vraies :

- ✓ Scattering (dispersion) : les attributs (méthodes) appartiennent à plus d'une classe.
- ✓ Tangling (enchevêtrement) : différentes méthodes d'une même classe contribuent à plus d'un cas d'utilisation.

Dans ce mémoire, nous allons proposer, par adaptation aussi, une deuxième technique inspirée de celle-ci en se basant sur l'analyse des scénarios d'exécution (extraits par analyse statique du code) au lieu des cas d'utilisations.

2.4.3 *Traitement du langage naturel*

Cette approche utilise le traitement du langage naturel à titre d'indice pour d'éventuelles préoccupations transverses. Elle est basée sur une technique NLP (*Natural Language Processing*) appelée chaînage lexical afin de trouver des groupes d'entités de code source connexes qui représentent des préoccupations transverses. La technique NLP prend en entrée une collection de mots et en sortie des chaînes de mots qui sont fortement liés. Afin de créer la chaîne, l'algorithme utilise *WordNet*, un catalogue de mots avec leurs sémantiques. Cet algorithme est appliqué sur les commentaires, les noms des méthodes, les noms des champs ainsi que les noms des classes d'un système donné. À la fin, une analyse

manuelle des chaînes résultantes est nécessaire pour l'identification des aspects candidats [16].

2.4.4 *Détection des méthodes uniques*

Gybels et Kellens proposent la détection des méthodes uniques [7]. Une méthode unique est définie comme « une méthode sans valeur de retour qui implémente un message qui n'est implémenté par aucune autre méthode ». Cette technique est appliquée selon les étapes suivantes :

- ✓ L'identification de toutes les méthodes uniques;
- ✓ Le tri de ces méthodes en fonction du nombre de fois qu'une méthode est appelée;
- ✓ Le filtrage des méthodes non pertinentes comme les accesseurs et les modificateurs (*gets et sets*);
- ✓ La vérification manuelle des méthodes résultantes afin de trouver les aspects candidats convenables.

Quelque soit la simplicité de cette méthode, les auteurs ont démontré l'applicabilité de leur technique en détectant des aspects typiques comme la notification et la gestion de la mémoire.

2.4.5 *La technique Fan-in*

Dans le chapitre cinq, nous allons appliquer cette technique pour des fins de comparaison. Pour cela nous décrivons, dans ce qui suit, cette technique en détail.

Marius Marin, Arie van Deursen et Leon Moonen ont défini le *Fan-in* d'une méthode *m*, comme le nombre des méthodes distinctes qui font appel à la méthode *m*. À cause du polymorphisme, un appel de méthode peut affecter le *Fan-in* de plusieurs autres méthodes. Autrement dit, un appel à une méthode *m* contribue au *Fan-in* de *m*, mais aussi au *Fan-in* de toutes les méthodes qui redéfinissent *m* [8]. Cela est illustré dans les deux figures suivantes :

```

interface A {
public void m();
}

class C1 extends B {
public void m() {}
}

class D {
void f1(A a) { a.m(); }
void f2(B b) { b.m(); }
void f3(C1 c) { c.m(); }
}

class B implements A {
public void m() {}
}

class C2 extends B {
public void m() { super.m(); }
}

```

Figure 3 Divers appels de la méthode polymorphe *m* [8].

Tableau 1 Fan-in des méthodes de la figure 3 [8].

méthodes	appels	Fan-in
A.m	{D.f1, D.f2, D.f3}	3
B.m	{D.f1, D.f2, D.f3, C2.m}	4
C1.m	{D.f1, D.f2, D.f3}	3
C2.m	{D.f1, D.f2}	2

D'après la figure 3, la méthode *B.m* est appelée dans *f2* et dans *m* de la sous classe *C2* (*super.m()*), cela implique que le *Fan-in* de *B.m* est égal à 2. Mais, puisque il existe un cas de polymorphisme dans la classe *D* à cause de la méthode polymorphe *m*, nous considérons alors que les deux appels de la méthode *m* dans *f1* et *f3* sont aussi des appels à la méthode *m* de la classe *B* comme il est illustré dans le tableau 1 ligne 2, cela augmente le *Fan-in* de

B.m qui devient égal à 4 [8]. Cette technique est très utile pour définir les méthodes dispersées dans le code source. Le calcul du *Fan-in* de chaque méthode est considéré comme une bonne mesure de l'importance et de la dispersion de la préoccupation découverte [9].

Cette technique est basée sur les étapes suivantes :

- ✓ Calculer le *Fan-in* de toutes les méthodes du système analysé.
- ✓ Filtrer le résultat : négliger les méthodes non pertinentes telles que les *gets* et les *sets* ainsi que les méthodes utilitaires comme *toString()*.
- ✓ Prendre en considération seulement les méthodes qui ont un *Fan-in* supérieur à un certain seuil.
- ✓ Analyser manuellement les méthodes restantes.

2.5 Classification des techniques

Les tableaux 2 et 3 comparent les techniques dont nous avons discutées [5]. Pour gagner un peu d'espace, nous avons abrégé les noms des techniques utilisées, telles que résumées par le tableau 2.

Tableau 2 La liste des techniques comparées.

Nom abrégé	Description
Appels récurrents	La technique d'analyse des appels récurrents et transverses basée sur les traces d'exécution
FCA	La technique d'analyse formelle de concepts basée sur les traces d'exécution
NLP	Traitement du langage naturel du code source (Natural language processing on source code)
Méthodes uniques	Détection des méthodes uniques
Fan-in	La technique Fan-in

Tableau 3 Classification des différentes techniques automatiques d'aspect mining.

Nom	Scattering	Tangling	Statique	Dynamique	Symptômes
Appels récurrents				X	Appels récurrents
FCA	X	X		X	Scattering/ Tangling
NLP	X		X		Nom
Méthodes uniques	X		X		Idiome
Fan-in	X		X		Grande dispersion

Comme nous pouvons le voir dans le tableau 3, la plupart des techniques sont basées sur une analyse statique du code source. Ces approches, exceptée la technique FCA, essayent d'identifier les aspects candidats caractérisés par des niveaux élevés de dispersion. Toutefois, *Ceccato* et *Tonella* ont pris en compte dans leur approche FCA les préoccupations transverses de type *tangling*, autrement dit, les préoccupations entremêlées dans le code source. À partir de cette classification, nous supposons qu'une liste de problèmes de recherche reste ouverte, ainsi qu'une liste de voies pour de futures recherches [5].

Chapitre 3 - La technique d'analyse des appels récurrents et transverses basée sur les scénarios

3.1 Introduction

Dans ce chapitre, nous présentons une adaptation de la technique d'analyse des appels récurrents basée sur les traces d'exécution proposées par Breu et Krinke [14]. Tel que décrit dans la section 2.4.1 du chapitre précédent, et contrairement à la technique originale qui est purement dynamique, la technique résultante (suite à l'adaptation) est basée sur une analyse statique des scénarios. Les scénarios d'exécution sont générés par analyse statique du code. Un autre avantage que procure notre approche est que la technique de Breu et Krink est une technique partielle, dans le sens où elle ne peut identifier que les aspects candidats impliqués durant l'exécution (selon les traces d'exécution). Notre approche, cependant, prend en compte tous les chemins d'exécution (générés par analyse statique). Cela nous permet de détecter davantage d'appels récurrents et transverses possibles.

3.2 Algorithme de la technique

L'idée principale derrière cette technique est d'observer le comportement d'exécution d'un système donné et d'extraire des informations à partir des chemins d'exécution. Ces informations peuvent nous aider à identifier les modes d'exécutions récurrents (suite de méthodes exécutées d'une façon répétitive et uniforme). Ces modes d'exécution sont considérés comme des aspects candidats.

Pour détecter ce genre de modes d'exécution récurrents, la technique que nous proposons est exécutée selon un algorithme comportant quatre phases :

- a) Générer les scénarios d'exécution (chemins d'exécution des méthodes).
- b) Classifier les relations d'exécution entre méthodes à partir des scénarios.
- c) Établir les contraintes des relations d'exécution (définition de l'ensemble des relations d'exécution vides ainsi que l'ensemble des relations d'exécution uniformes).
- d) Identifier les appels récurrents et transverses.

3.2.1 Génération des scénarios

Nous avons utilisé un générateur de chemins d'exécution compactés, implémenté dans le cadre des travaux de recherche de Daniel St-Yves [17]. Ce générateur a été développé pour supporter une méthode d'analyse de l'impact des changements basée sur le contrôle. La génération des chemins d'exécution est organisée en deux étapes :

3.2.1.1 Analyse du code source et génération des graphes de contrôle réduits aux appels

L'objectif de cette étape consiste à effectuer une analyse statique du code source d'un programme afin de construire une synthèse des algorithmes des différentes méthodes. Ces algorithmes permettent la construction des graphes de contrôle réduits aux appels. La figure 4.2 donne la synthèse de l'algorithme de la méthode $M()$ définie dans la figure 4.1. Les instructions ne contenant pas des appels de méthodes ont été supprimées (figure 4.2). La figure 4.3 donne le graphe de contrôle correspondant.

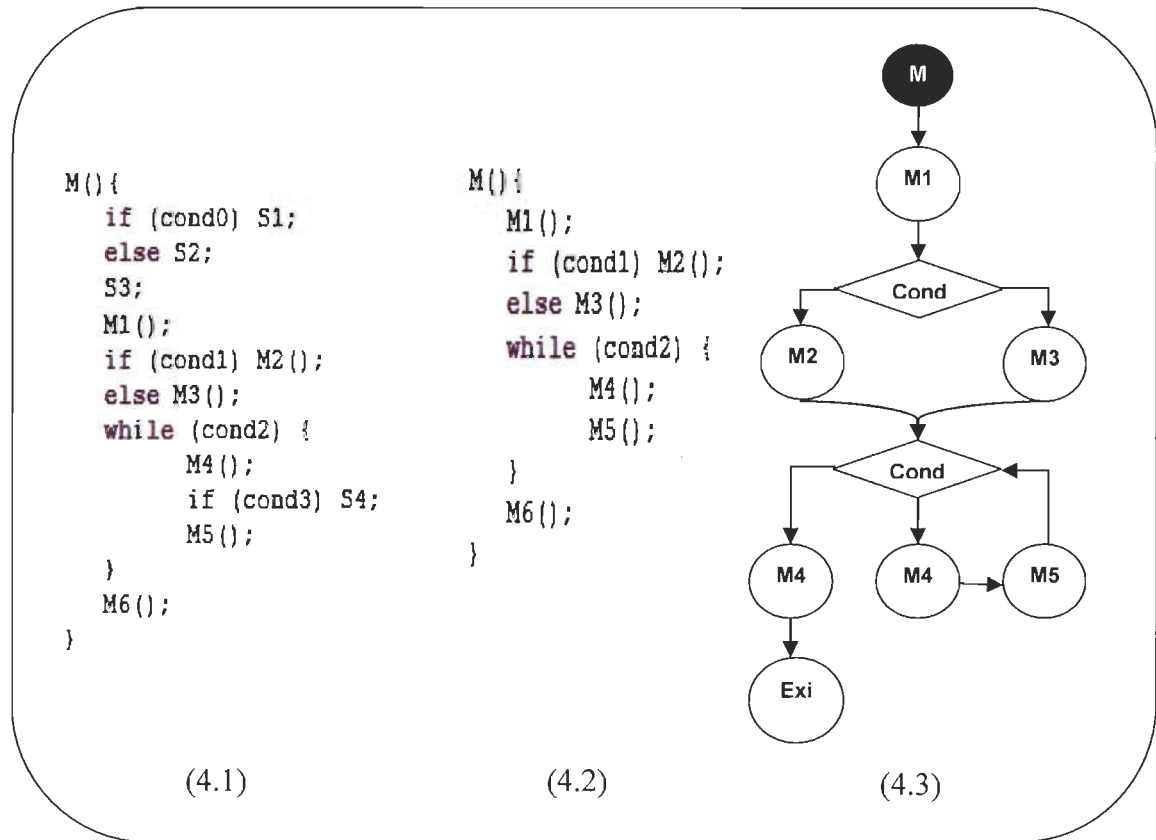


Figure 4 Construction des graphes de contrôle réduit aux appels.

3.2.1.2 Génération des chemins d'exécution

Cette étape consiste à analyser les graphes de contrôle réduits obtenus lors de l'étape précédente pour la génération de chemins de contrôle compactés. À partir de ces chemins, nous pouvons générer l'ensemble des chemins de contrôle réduits aux appels, en éliminant les chemins infaisables [17]. La figure 5 illustre cela :


```

M() {
  M1();
  if (cond1) M2();
  else M3();
  while (cond2) {
    M4();
    M5();
  }
  M6();
}

M2() {
  M7();
  if (cond3) M2();
}

M3() {
  M8();
}

M6() {
  if (cond4) M8();
  M10();
}

M8() {
  M9();
}

```

Figure 5 Contrôle réduit aux appels de plusieurs méthodes.

La figure 5 donne la synthèse (contrôle réduit aux appels) de plusieurs méthodes que nous considérons pour illustrer l'approche.

M:M1,(M2,M3),{M4,M5},M6
 M2:M7,[M8]
 M3:M8
 M6:[M8],M10
 M8:M9

Figure 6 Chemins d'appels compactés.

M:M1,M2,M6
 M:M1,M2,M4,M5,M6
 M:M1,M2,M4,M5, M4,M5,M6
 M:M1,M3,M6
 M:M1,M3,M4,M5,M6
 M:M1,M3,M4,M5, M4,M5,M6
 M2:M7
 M2:M7,M8
 M3:M8
 M6:M10
 M6:M8,M10
 M8:M9

Figure 7 Chemins d'exécutions possibles.

La figure 6 présente les chemins de contrôle compactés correspondants aux méthodes de la figure 5. Nous utilisons plusieurs notations pour exprimer le contrôle dans les séquences d'appels. Les accolades {} expriment l'itération (while, for), la séquence entre

accolades peut être exécutée 0, une ou plusieurs fois. Les séquences entre parenthèses () expriment l'alternative (if, else if, else), une de ces séquences est exécutée. La séquence entre crochet [] exprime le fait qu'elle peut être exécutée comme elle ne peut pas l'être. La figure 7 représente une partie des chemins d'exécution possibles qui peuvent être déduits des chemins de contrôle de la figure 6 qui sont automatiquement générés par analyse statique du code source. Ceci représente un avantage important relativement aux approches dynamiques qui tentent de les obtenir par instrumentation du code et compactage des traces d'exécution [17].

3.2.2 *Classification des relations d'exécution*

Les relations d'exécution dans notre approche sont extraites à partir des chemins d'exécution. Nous nous intéressons aux appels de méthodes. Les fonctionnalités implémentées dans les systèmes analysés sont encapsulées dans les méthodes. Il existe quatre types de relations d'exécutions :

- ✓ Outside-before
- ✓ Outside-after
- ✓ Inside-first
- ✓ Inside-last

Pour bien comprendre ces quatre relations d'exécution, nous considérons l'exemple suivant :

```

void m1() {
    a();
    b();
    a();
}

void m2 (){
    a();
    if (...){
        b();
        a();
    }
}

void m3{
    a();
    c();
}

```

Figure 8 Exemple pour illustrer les relations d'exécution.

- La première relation est *outside-before* ou avant-extérieur. Dans l'exemple, nous remarquons dans la méthodes m1() que la méthodes a() est appelée avant la méthode b(), nous pourrions donc dire qu'il existe une relation *outside-before* entre a() et b() et nous notons cette relation : $a() > b()$.
- De la même manière, nous définissons *outside-after* ou après-extérieur. Nous remarquons dans l'exemple précédent que la méthode c() est exécutée après la méthode a() dans m3(), il existe alors une relation d'exécution de type *outside-after* entre c() et a() notée de cette façon : $c() < a()$.
- La troisième relation est *inside-first* ou première-intérieur. Dans notre exemple, a() est la première méthode appelée à l'intérieur de la méthode m2(), cela signifie qu'il y a une relation *inside-first* entre a() et m2(). Notons cette relation par $a() \# m2()$.

- La dernière relation est *inside-last* ou dernière-intérieur. Dans la déclaration de la méthode $m3$, $c()$ est la dernière méthode appelée à l'intérieur de $m3()$, il y a donc une relation *inside-last* entre $c()$ et $m3()$: $c() @ m3()$.

Nous continuons sur le même exemple. Nous allons définir l'ensemble des quatre relations d'exécution. Nous supposons que :

- $S_>$: L'ensemble des relations d'exécution *outside-before*.
- $S_<$: L'ensemble des relations d'exécution *outside-after*.
- $S_\#$: L'ensemble des relations d'exécution *inside-first*.
- $S_@$: L'ensemble des relations d'exécution *inside-last*.

Si nous construisons ces ensembles de relations d'exécution à partir de l'exemple précédent nous obtiendrons :

- $S_> = \{a() > b(), b() > a(), a() > c()\}$.
- $S_< = \{a() < b(), b() < a(), a() < c()\}$.
- $S_\# = \{a() \# m1(), a() \# m2(), a() \# m3()\}$.
- $S_@ = \{a() @ m1(), a() @ m2(), c() @ m3()\}$.

Dans l'étape suivante, nous discutons les différentes contraintes des relations d'exécution.

3.2.3 Les contraintes des relations d'exécution

Étape 1 : Le codage des méthodes vides \mathbb{E}

Nous avons besoin de définir les relations d'exécution vides. Ces relations sont appelées aussi *Epsilon relations*. Pour cela nous allons :

- ✓ Identifier les méthodes ne contenant pas d'appels de méthodes. Si nous supposons que la méthode $a()$ ne contient aucun appel de méthodes, alors nous devrions définir deux relations d'exécution vides : $\mathbb{E} \# a()$ (aucun appel exécuté en premier dans $a()$) et $\mathbb{E} @ a()$ (aucun appel exécuté en dernier dans $a()$).
- ✓ Définir des relations d'exécution vides pour les deux types de relations d'exécution *inside-first* et *inside-last* : dans notre exemple, la relation d'exécution *inside-first* $a() \# m1()$ ($a()$ est la première méthode appelée dans $m1()$) génère la relation d'exécution vide $\mathbb{E} > a()$ (aucun appel de méthode avant $a()$), la même chose pour la relation d'exécution *inside-last* $c() @ m3()$ ($c()$ est la dernière méthode appelée dans $m3()$) génère la relation $\mathbb{E} < c()$ (aucun appel de méthode après $c()$).

Notons E l'ensemble des relations d'exécution vides et S l'ensemble de toutes les relations d'exécution de la figure 8 :

$$E = \{ \mathbb{E} > a(), \mathbb{E} < a(), \mathbb{E} < c(), \mathbb{E} \# a(), \mathbb{E} @ a(), \mathbb{E} \# b(), \mathbb{E} @ b(), \mathbb{E} \# c(), \mathbb{E} @ c() \}$$

$$S = S_{>} \cup S_{<} \cup S_{\#} \cup S_{@} \cup E$$

Étape 2 : Les relations d'exécution uniformes

Notons M un ensemble de méthodes et R une relation d'exécution entre deux méthodes où $M = \{u, v, w\}$ et $R \in \{<, >, \#, @\}$. Une relation d'exécution $u R v$ est dite uniforme s'il n'y a pas une autre méthode w différente de u qui a la même relation R avec la méthode v . Formellement, une relation d'exécution $u R v$ est dite uniforme si $\forall w R v : u = w$ avec $u, v, w \in M \cup \{\text{£}\}$ où $\{\text{£}\}$ est l'ensemble des relations vides.

Dans la figure 8, la relation $a() > b()$ est une relation uniforme, car il n'existe pas une autre méthode différente de $a()$ exécutée avant $b()$ sauf $a()$. Par contre, la relation $b() < a()$ n'est pas uniforme, car il ya une autre méthode $c()$ exécutée après $a()$ ($c() < a()$).

Notons \hat{S} l'ensemble des relations d'exécution uniformes non vides de S :

$$\hat{S} = \{ a() > b(), a() > c(), a() < b(), a() \# m1(), a() \# m2(), a() \# m3(), \\ a() @ m1(), a() @ m2(), c() @ m3() \}$$

L'ajout des relations d'exécutions vides rend certaines relations d'exécution non uniformes telles que :

- Les méthodes redéfinies dans le code source une fois avec un appel de méthodes et une autre fois sans appel de méthodes. Soit une méthode v redéfinie une fois avec un appel de méthode : $v()\{u()...u'()\}$. Nous avons donc deux relations d'exécution $u \# v$ (u la première méthode appelée dans v) et $u' @ v$ (u' la dernière méthode exécutée dans v). v est aussi déclarée sans un appel de méthode d'où $\text{£} \# v$ et $\text{£} @ v$. En conséquence, u et £ ont la même relation *Inside-first* avec v d'où les deux relations ne sont pas uniformes. La même chose est dite pour les deux relations $u' @ v$ et $\text{£} @ v$.

- Une méthode appelée une fois après une autre méthode et une autre fois elle est appelée la première dans une méthode. Nous supposons v appelée après u ($u > v$) et aussi elle est la première méthode appelée dans u' d'où $\mathcal{E} > v$. Des deux relations $u > v$ et $\mathcal{E} > v$, nous constatons que u et \mathcal{E} ont la même relation ($>$) avec v , elles ne sont donc pas uniformes.
- Une méthode appelée une fois avant une autre méthode et une autre fois elle est appelée la dernière dans une méthode. Nous supposons v appelée avant u ($u < v$) et aussi elle est la dernière méthode appelée dans u' d'où $\mathcal{E} < v$. Des deux relations $u < v$ et $\mathcal{E} < v$, nous constatons que u et \mathcal{E} ont la même relation ($<$) avec v , elles ne sont donc pas uniformes.

Les relations d'exécution vides permettent d'éliminer certaines relations, cela implique la diminution du nombre d'aspect candidats détectés.

Remarque :

Les relations d'exécution non uniformes de l'exemple précédent sont $b() > a()$, $\mathcal{E} > a()$, $b() < a()$, $c() < a()$, $\mathcal{E} < a()$.

3.2.4 L'identification des appels récurrents et transverses

La définition formelle d'une relation d'exécution transverse est : $s = u R v \in \hat{S}^R$ tel que $R \in \{<, >, \#, @\}$, est dite transverse si $\exists u R w \in \hat{S}^R : v \neq w$ avec $u, v, w \in M$ où M est l'ensemble des méthodes et \hat{S}^R l'ensemble des relations d'exécution uniformes non vide. En d'autres termes, si nous avons une relation d'exécution uniforme $u R v$, cette relation est dite transverse s'il y a au moins une autre méthode w différente de v tel que u a une relation avec w ($u R w$). Notons T l'ensemble des relations d'exécution uniformes et transverses :

$$T = \{a() > b(), a() > c(), a() \# m1(), a() \# m2(), a() \# m3(), a() @ m1(), a() @ m2()\}$$

Remarque :

$a() < b()$ et $c() @ m3()$ sont des relations d'exécution uniformes non transverses.

Chapitre 4 - La technique d'Analyse Formelle de Concepts (FCA) basée sur les scénarios

4.1 Introduction

Dans ce chapitre, nous décrivons la technique d'Analyse Formelle des Concepts (FCA) basée sur les traces d'exécution (son adaptation à notre contexte). À partir de la définition des concepts et des treillis de concepts, la technique permet de détecter les portions de code pouvant correspondre à des aspects candidats. Ces portions de code peuvent être une hiérarchie de classes, des classes ou des méthodes [18].

L'objectif de notre étude est d'évaluer si cette technique (plus exactement son adaptation) pourrait être utilisée dans le cadre d'une analyse statique du code source d'une application et détecter d'une façon automatique (par simple analyse statique du code) la présence de préoccupations transverses. L'adaptation effectuée de la technique permet une analyse formelle des concepts par analyse statique du code, basée sur les chemins générés, suivant le même principe que la technique décrite dans le chapitre 3. Les concepts sont des groupes maximaux d'objets qui ont des attributs en commun. Dans notre cas, les objets sont des scénarios et les attributs sont des méthodes. Nous considérons certaines méthodes comme des aspects candidats si les deux contraintes suivantes sont vérifiées :

- ✓ Scattering : méthodes appartenant à plus d'une classe,
- ✓ Tangling : différentes méthodes d'une même classe contribuant dans plus d'un scénario.

Dans ce qui suit, nous définissons une configuration spécifique de l'algorithme FCA appliquée sur des scénarios pour détecter les aspects candidats.

4.2 Algorithme de la technique

Cette technique est assez simple. À partir d'un ensemble (potentiellement large) d'objets et d'attributs, la technique FCA détermine les groupes maximaux d'objets qui ont des attributs en commun. Ces groupes maximaux d'objets sont appelés des concepts. Chaque concept se compose d'un ensemble d'objets ayant un ou plusieurs attributs en commun. Cette technique est appliquée selon un algorithme comportant les quatre phases suivantes :

- a) Génération des scénarios (chemins d'exécution des méthodes),
- b) Définition du contexte formel et la liste des concepts à partir du contexte formel,
- c) Définition de treillis de concepts,
- d) Identification des préoccupations transverses.

4.2.1 Génération des scénarios

Nous utilisons le même générateur de chemins d'exécution décrit dans la section 3.2.1 du chapitre précédent en passant par les deux étapes suivantes :

- Analyse du code source et génération des graphes de contrôle réduits aux appels,
- Génération des chemins d'exécution.

4.2.2 Contextes Formels

Nous appelons un contexte formel la relation entre des objets et leurs attributs. Cette relation peut être représentée par un tableau. Les lignes sont l'ensemble des objets, et les colonnes sont l'ensemble des attributs et la relation entre eux est représentée par des croix (voir figure ci-dessous) [19].

Tableau 4 Contexte formel sur certains animaux.

	Carnivore	Végétarien	Domestique	Oiseau	Mammifère	Sauvage
Lion	X				X	X
Aigle	X			X		X
Zèbre		X			X	X
Cygne		X		X		X
Lapin		X	X		X	

Dans l'exemple précédent, les objets sont des animaux et les attributs décrivent si l'objet est un carnivore ou un végétarien, s'il est domestique ou sauvage, et si c'est un oiseau ou un mammifère. À partir de n'importe quel ensemble d'objets, nous pouvons identifier tous les attributs qu'ils ont en commun. Dans le tableau 4, le seul attribut commun entre le cygne et le lapin, est qu'ils sont végétariens. Cependant, ils ne sont pas les seuls animaux végétariens, le Zèbre est aussi végétarien. Cela veut dire que « végétarien » est le seul attribut commun entre ces trois animaux. Donc, nous disons que l'ensemble

{Zèbre, Cygne et lapin} est le groupe maximal d'objets ayant l'attribut « végétarien » en commun.

4.2.3 Concepts

Un concept C décrit un ensemble d'objets $O = \{o_1, o_2, o_3 \dots o_n\}$ qui ont un ensemble d'attributs $A = \{a_1, a_2, a_3 \dots a_m\}$ en commun. Notons :

$$C = (\{o_1, o_2, o_3 \dots o_n\}, \{a_1, a_2, a_3 \dots a_m\});$$

À partir de l'exemple précédent, nous pouvons définir un concept C pour les trois animaux végétariens : $C = (\{Zèbre, Cygne, Lapin\}, \{Végétarien\})$ où {Zèbre, Cygne, Lapin} est un ensemble d'objet qui ont l'attribut {Végétarien} en commun.

Nous appelons l'ensemble d'objets O l'extension du concept C et l'ensemble d'attributs A l'intension de C .

Nous disons qu'un concept $C1 = (O1, A1)$ spécialise $C2 = (O2, A2)$ si l'extension de $C1$ est incluse dans l'extension de $C2$ et inversement l'intension de $C2$ est incluse dans l'intension de $C1$, autrement dit, $O1 \subset O2$ et $A2 \subset A1$ [19].

4.2.4 Treillis de concepts

Dans cette phase nous traçons le treillis de concepts.

Nous considérons le contexte formel (O, A, R) tel que :

✓ O : Ensemble d'objets $O = \{o_1, o_2, o_3 \dots o_n\}$

✓ A : Ensemble d'attributs $A = \{a_1, a_2, a_3 \dots a_m\}$

R : Relation binaire entre un objet o dans O et un attribut a dans A .

Tableau 5 Contexte formel d'un ensemble d'objets O et d'attributs A [15].

	a1	a 2	a 3	a 4
o1	X	X	X	
o 2	X		X	X
o 3		X	X	

Un treillis de concepts L représente chaque concept par un nœud et chaque couple de concepts $(C1, C2)$ par un arc où C1 spécialise C2. Le treillis de concepts L contient un nœud top en haut, son extension contient tous les objets et un nœud bot en bas, son intention contient tous les attributs [15].

Remarque :

S'il n'y a pas un ensemble d'objets qui ont en commun l'ensemble de tous les attributs alors l'extension du bot est vide. Même chose pour le concept top, s'il n'y a pas d'attributs en commun entre l'ensemble de tous les objets, donc l'intension du top est vide.

Voici l'ensemble des concepts du tableau 5 ainsi que le treillis de concepts :

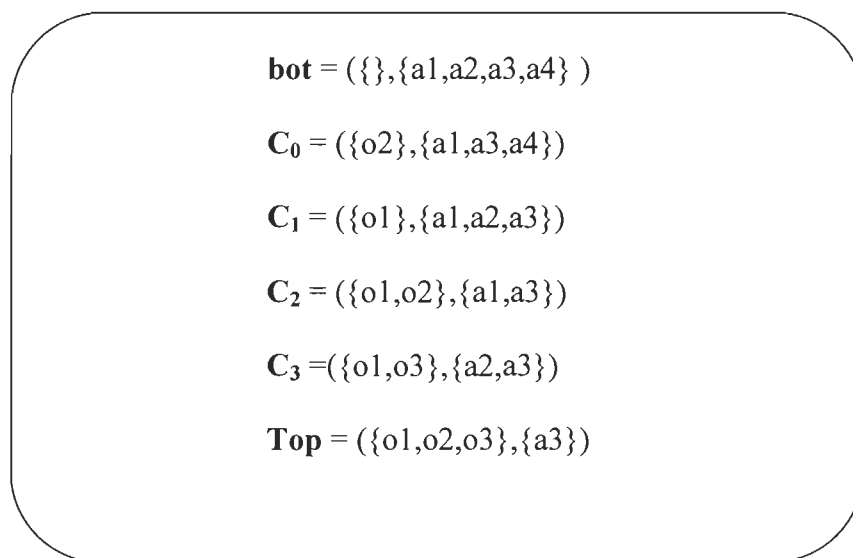


Figure 9 L'ensembles des concepts du contexte formel présenté dans le tableau 5.

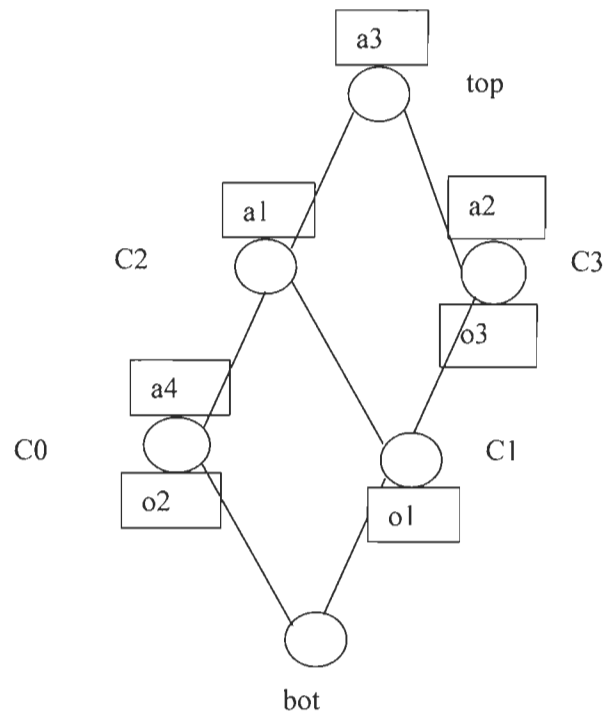


Figure 10 Exemple de treillis de concepts [15].

4.2.5 Identification des preoccupations transverses

Les concepts qui sont définis et présentés dans les phases précédentes sont analysés et classés selon certains critères. Lorsque nous traitons des concepts contenant des méthodes, par exemple, nous pouvons les classer selon les catégories dans lesquelles elles sont définies.

Nous pouvons, par exemple, distinguer entre les concepts contenant des méthodes qui sont toutes définies dans la même classe, les concepts contenant toutes les méthodes définies dans une hiérarchie particulière, ou contenant des méthodes transverses qui sont définies dans les différentes classes qui ne sont pas liées par héritage. Bien sûr, d'autres critères peuvent être définis selon la technique appliquée et selon nos objectifs [18].

Dans notre cas, nous traitons des concepts contenant des scénarios comme des objets et des attributs comme des méthodes. De cette façon, nous pouvons identifier les méthodes appartenant à plus d'une classe et les différentes méthodes d'une même classe contribuant dans plus d'un scénario. Ce sont les méthodes que nous considérons comme des aspects candidats.

4.3 Application de la technique sur un exemple

Prenons un exemple de recherche dans un arbre binaire composé des deux classes suivantes [15] :

Tableau 6 Les deux classes Arbre_Binaire et Nœud.

Classe : Arbre_Binaire

Racine : Nœud

Arbre_Binaire()

Inserer(n : Nœud)

Chercher(x : information) : boolean

Classe : Nœud

Droit : Nœud

Gauche : Nœud

Y : information

Nœud(x : information)

Inserer(z : Nœud)

Chercher(x : information) : Nœud

Ces principales fonctionnalités sont l'insertion de nœuds et la recherche d'informations à l'intérieur de l'arbre. Pour simplifier, nous appliquons la technique sans passer par les chemins d'exécution. Une autre raison est que dans le prochain chapitre nous évaluons cette technique sur des exemples concrets. Voici donc les deux cas d'utilisation de l'exemple :

Tableau 7 Relation entre cas d'utilisations et les appels de méthodes [15].

Recherche d'informations
Arbre_Binaire. Arbre_Binaire() Arbre_Binaire.Chercher(information) Nœud.Chercher(information)
Insertion des nœuds
Arbre_Binaire. Arbre_Binaire() Arbre_Binaire.Inserer(Nœud) Nœud.Inserer(Nœud) Nœud.Nœud(information)

Le contexte formel est tracé en considérant les deux cas d'utilisation comme des objets et les méthodes appelées comme des attributs :

Tableau 8 Contexte formel de l'arbre binaire.

	Arbre_Binaire. Arbre_Binaire ()	Arbre_Binaire. Chercher (information)	Arbre_Binaire. Insérer(Nœud)	Nœud. Chercher (information)	Nœud. Insérer (Nœud)	Nœud. Nœud (information)
Recherche des nœuds	X	X		X		
Insertion des nœuds	X		X		X	X

Les concepts définis à partir de ce contexte formel sont :

$bot = (\{\}, M)$ où M l'ensemble de toutes les methods appelées

$C0 = (\{Recherche\ des\ nœuds\}, \{Arbre_Binaire.Chercher(information), Nœud.Chercher(information)\})$

$C1 = (\{Insertion\ des\ nœuds\}, \{Arbre_Binaire.Insérer(Nœud), Nœud.Insérer(Nœud), Nœud.Nœud(information)\})$

$top = (\{Recherche\ des\ nœuds, Insertion\ des\ nœuds\}, \{Arbre_Binaire.Arbre_Binaire()\})$

Figure 11 Les concepts définis du tableau 8.

Nous remarquons qu'il n'y a pas de cas d'utilisation qui inclut tous les attributs, l'extension (l'ensemble des objets) du concept bot est donc vide. Par contre, les deux cas d'utilisation ont *Arbre_Binaire.Arbre_Binaire()* comme attribut en commun, ce qui explique l'extension du concept top qui inclut les deux cas d'utilisation. Voici le treillis de concepts obtenu :

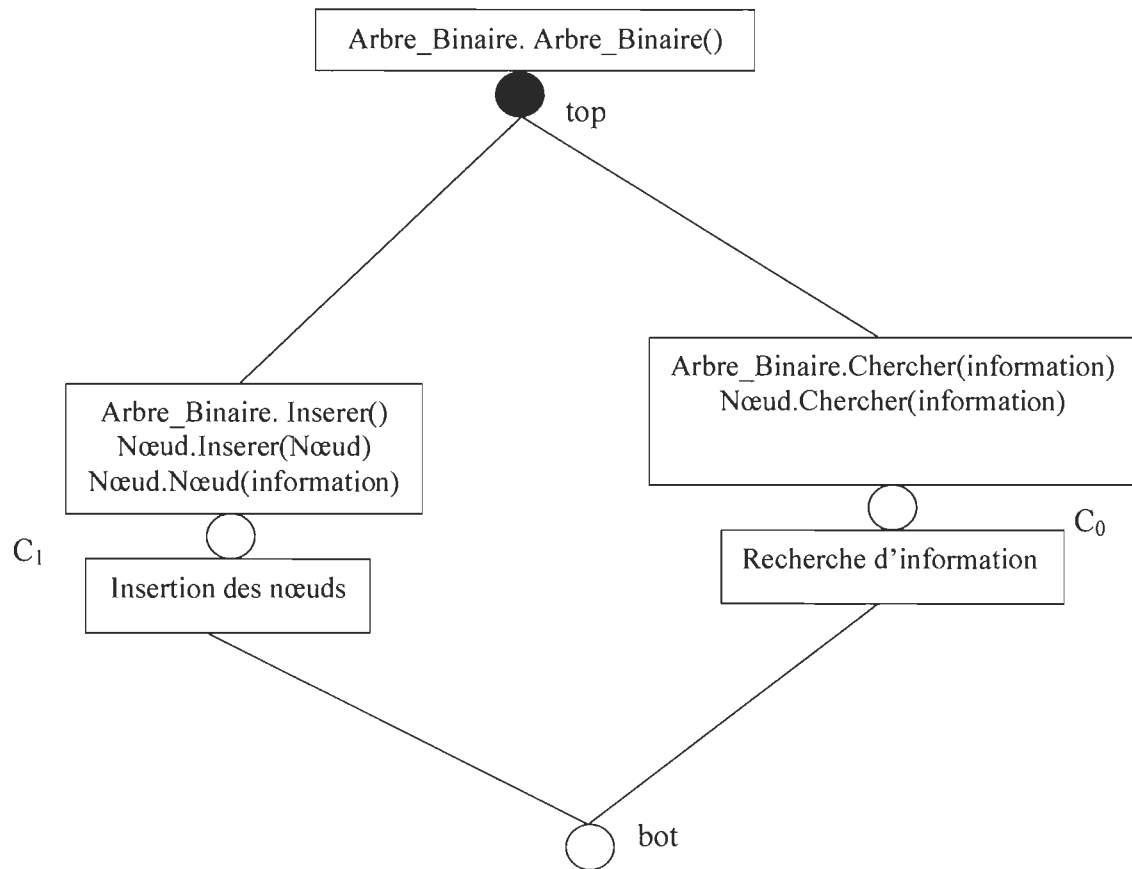


Figure 12 Treillis de concept de l'arbre binaire [15].

Ce treillis de concepts indique que les deux fonctionnalités « insertion des nœuds » et « recherche d'information » sont des préoccupations transverses. En fait, les deux cas d'utilisation des concepts spécifiques sont étiquetés par des méthodes qui appartiennent à plus d'une classe. Ce sont les deux méthodes *Insérer* et *Chercher*. La deuxième cause est le fait que chaque classe contribue à plus d'une fonctionnalité. En d'autres termes, les deux fonctionnalités sont entremêlées dans la classe *Arbre_Binaire* et dans la classe *Nœud* [15].

Ce résultat peut être interprété comme le fait que ces deux classes ne sont pas très cohésives. Il serait possible de séparer chacune de ces fonctions transverses et de les localiser à l'intérieur d'un module bien séparé (soit une nouvelle sous-classe ou un *aspect*). Par exemple, une classe de base *Arbre_Binaire* pourrait être héritée par une sous-classe *ArbreBinaireRecherche* qui ajoute la fonctionnalité de recherche d'information [15].

Chapitre 5 - Évaluation empirique

5.1 Introduction

Cette section présente une étude expérimentale au cours de laquelle nous avons considéré, dans un premier temps, trois exemples de classes écrites en java. Elles ont été soigneusement sélectionnées. Ces exemples ont, en effet, déjà été discutés dans la littérature spécialisée dans les domaines de la maintenance, de la restructuration et de l'*aspect mining*. À la fin de ce chapitre, nous présenterons une étude empirique que nous avons effectuée sur un benchmark très utilisé dans la littérature « *JHotDraw* ». Il représente un système réel d'envergure. Les résultats obtenus sont analysés en utilisant un outil d'analyse statistiques *XSTAT*. Cet outil intègre plusieurs utilitaires statistiques et d'analyse de données.

Nous commençons tout d'abord par les exemples. Le premier exemple sera décrit en détail. Pour les deux autres, nous présenterons (en suivant la même démarche) seulement les éléments de discussion les plus pertinents sans toutefois nous étaler sur les détails.

5.2 Exemple 1 (Design Pattern Observer)

5.2.1 Présentation de l'exemple

Le but du modèle Observateur (Pattern Observer) est de définir une à plusieurs dépendances entre les objets de telle sorte que lorsqu'un objet change d'état, toutes ses dépendances sont notifiées et modifiées automatiquement. Prenons comme exemple (figure 13) un schéma du pattern observateur (données observables visualisées sur un écran par un observateur) [20]. Nous souhaitons que la modification d'un objet observable (appel à l'une des deux méthodes préfixées par *set*) modifie la présentation graphique affichée sur l'écran. Afin d'éviter l'utilisation de thread ou encore modifier l'affichage directement depuis chacune des méthodes *sets*, ce qui entraîne des problèmes en termes de réutilisation et de maintenance, il suffit d'utiliser le patron de conception observateur. Le principe est que chaque classe observable (sujet) comporte une liste d'observateurs. Ainsi, à l'aide d'une méthode de notification l'ensemble des observateurs est prévenu.

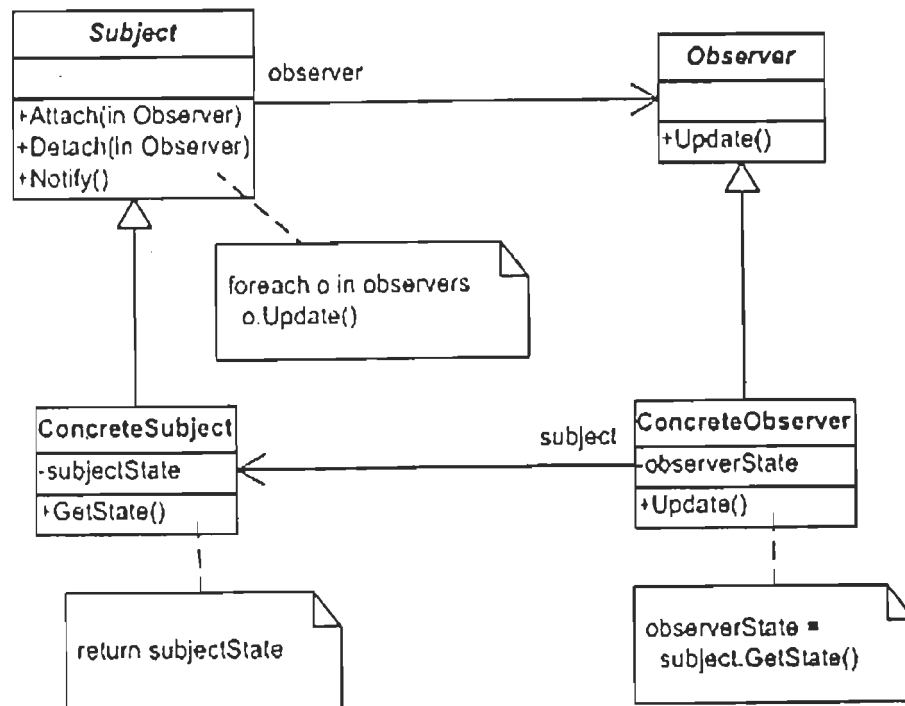


Figure 13 Diagramme UML du pattern Observer [20].

Nous donnons un exemple qui décrit un modèle observateur écrit en Java. Cet exemple contient quatre classes : *Point*, *GuiElement*, *Observer* et la classe principale *Main*. Ces classes sont présentées en détail dans l'annexe A. Dans la figure 14 qui suit, nous constatons que le code comporte des méthodes dispersées comme *add(observer)*, *remove(observer)* et *iterator()* dans les deux classes *Point* et *GuiElement* (code étiqueté C). Nous observons aussi l'enchevêtrement (*tangling*) du code dans la même classe (*Point*), entre sa fonctionnalité principale (code étiqueté A) mixé avec le code relatif au mécanisme de notification de l'observateur (code étiqueté B, C et D).

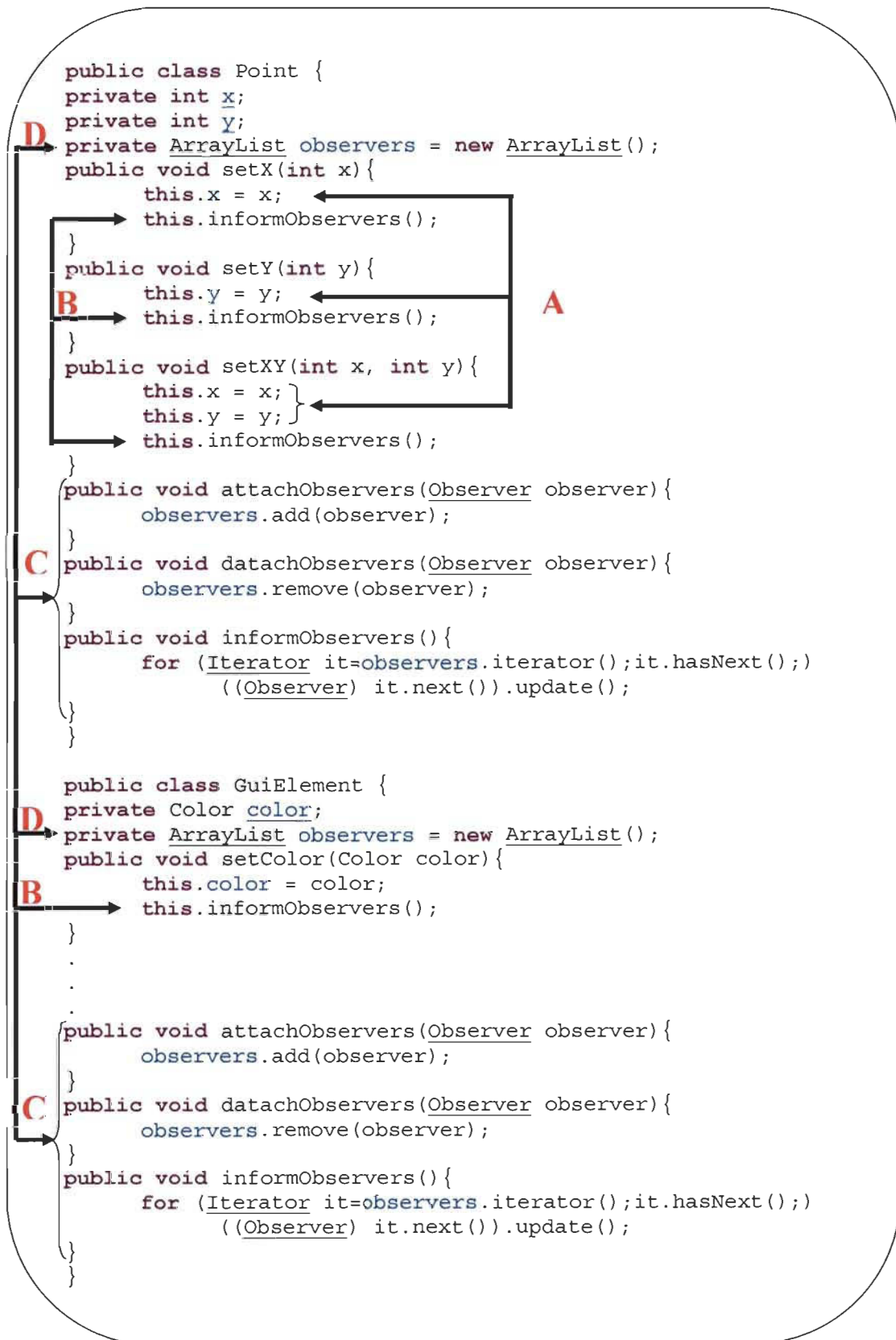


Figure 14 Code source du modèle observateur.

Dans ce qui suit, nous allons appliquer les deux approches décrites dans les précédents chapitres sur cet exemple et nous discuterons les résultats obtenus.

5.2.2 Évaluation de la technique d'analyse des appels récurrents et transverses basée sur les scénarios

Pour appliquer la technique, nous allons commencer tout d'abord par extraire les chemins d'exécution. Nous citons toutes les méthodes déclarées dans les quatre classes avec leurs appels de méthodes. Voici ce que nous obtenons :

```
GuiElement.informObservers:AbstractList.iterator,Iterator.hasNext,
{Observer.update,Iterator.next}
Point.attachObservers:ArrayList.add
Observer.update:
Point.detachObservers: ArrayList.remove
Point.setXY:Point.informObservers
Point.informObservers:AbstractList.iterator,Iterator.hasNext,{Ob
server.update,Iterator.next}
Point.setX: Point.informObservers
GuiElement.detachObservers: ArrayList.remove
GuiElement.setColor: GuiElement.informObservers
Point.setY: Point.informObservers
GuiElement.attachObservers: ArrayList.add
Main.main: Point.Point,Point.setX, GuiElement.informObservers
```

Figure 15 La liste des chemins d'exécutions de l'exemple 1.

L'étape suivante consiste à identifier les quatre types de relations d'exécution à partir des chemins d'exécution précédents. Pour cela, la technique consiste à chercher, chemin par chemin, les méthodes qui ont une relation de type *Inside-First* (#), ensuite la deuxième relation et ainsi de suite jusqu'à ce que nous obtenions toutes les relations d'exécution possibles. À titre d'exemple, pour trouver la relation *Inside-First* dans le premier chemin, il faut chercher quelle méthode est appelée en premier. Nous voyons bien que c'est la méthode *AbstractList.iterator*. Donc, nous avons une relation de type *Inside-First* entre

AbstractList.iterator

et

*GuiElement.informObservers**AbstractList.iterator#GuiElement.informObservers*). Voici la liste de toutes les relations

d'exécution :

```

inside-first :
AbstractList.iterator#GuiElement.informObservers
ArrayList.add#Point.attachObservers
ArrayList.remove#Point.detachObservers
Point.informObservers#Point.setXY
AbstractList.iterator#Point.informObservers
Point.informObservers#Point.setX
ArrayList.remove#GuiElement.detachObservers
GuiElement.informObservers#GuiElement.setColor
Point.informObservers#Point.setY
ArrayList.add#GuiElement.attachObservers
Inside-last :
Iterator.hasNext@GuiElement.informObservers
Iterator.next@GuiElement.informObservers
ArrayList.add@Point.attachObservers
ArrayList.remove@Point.detachObservers
Point.informObservers@Point.setXY
Iterator.hasNext@Point.informObservers
Iterator.next@Point.informObservers
Point.informObservers@Point.setX
ArrayList.remove@GuiElement.detachObservers
GuiElement.informObservers@GuiElement.setColor
Point.informObservers@Point.setY
ArrayList.add@GuiElement.attachObservers
Outside-before :
Iterator.next>Observer.update
AbstractList.iterator>Iterator.hasNext
Iterator.hasNext>Observer.update
Observer.update>Iterator.next
Point.Point>Point.setX
Point.setX>Point.setY
Point.setY>Point.Point
Point.Point>Point.setXY
Point.setXY>Point.setY
Outside-after :
Observer.update<Iterator.next
Iterator.hasNext<AbstractList.iterator
Observer.update<Iterator.hasNext
Iterator.next<Observer.update
Point.setX<Point.Point
Point.setY<Point.setX
Point.Point<Point.setY
Point.setXY<Point.Point
Point.setY<Point.setXY

```

Figure 16 Les relations d'exécutions de l'exemple 1.

Ensuite, et avant d'identifier les relations d'exécution uniformes, nous avons besoin de déterminer les relations vides ou ce qu'on appelle les *epsilon relations* :

```

inside-first:
vide#Observer.update:
Inside-last :
vide@Observer.update:
Outside-before :
vide>ArrayList.add
vide>ArrayList.remove
vide>AbstractList.iterator
vide>Point.informObservers
vide>GuiElement.informObservers
Outside-after :
Vide<Iterator.hasNext
Vide<Iterator.next
Vide<ArrayList.add
Vide<ArrayList.remove
Vide<Point.informObservers
Vide<GuiElement.informObservers

```

Figure 17 Les relations vides de l'exemple 1.

Comme nous l'avons discuté auparavant, une relation d'exécution est dite uniforme si et seulement si il n'y a pas deux méthodes différentes ou plus qui ont une même relation avec la même méthode. Pour cela, nous allons identifier les relations uniformes et éliminer les relations non uniformes. La figure 18 présente les relations d'exécution uniformes. Les relations soulignées ne sont pas uniformes.

```

inside-first :
AbstractList.iterator#GuiElement.informObservers
ArrayList.add#Point.attachObservers
ArrayList.remove#Point.detachObservers
Point.informObservers#Point.setXY
AbstractList.iterator#Point.informObservers
Point.informObservers#Point.setX
ArrayList.remove#GuiElement.detachObservers
GuiElement.informObservers#GuiElement.setColor
Point.informObservers#Point.setY
ArrayList.add#GuiElement.attachObservers
Inside-last :
Iterator.hasNext@GuiElement.informObservers
Iterator.next@GuiElement.informObservers
ArrayList.add@Point.attachObservers
ArrayList.remove@Point.detachObservers
Point.informObservers@Point.setXY
Iterator.hasNext@Point.informObservers
Iterator.next@Point.informObservers
Point.informObservers@Point.setX
ArrayList.remove@GuiElement.detachObservers
GuiElement.informObservers@GuiElement.setColor
Point.informObservers@Point.setY
ArrayList.add@GuiElement.attachObservers
Outside-before :
Iterator.next>Observer.update
AbstractList.iterator>Iterator.hasNext
Iterator.hasNext>Observer.update
Observer.update>Iterator.next
Point.Point>Point.setX
Point.setX>Point.setY
Point.setY>Point.Point
Point.Point>Point.setXY
Point.setXY>Point.setY
Outside-after :
Observer.update<Iterator.next
Iterator.hasNext<AbstractList.iterator
Observer.update<Iterator.hasNext
Iterator.next<Observer.update
Point.setX<Point.Point
Point.setY<Point.setX
Point.Point<Point.setY
Point.setXY<Point.Point
Point.setY<Point.setXY

```

Figure 18 Les relations d'exécution uniformes de l'exemple 1.

Dans l'ensemble des relations d'exécution précédent, nous avons des méthodes différentes qui ont la même relation avec la même méthode, comme les deux méthodes *Iterator.hasNext* et *Iterator.next* qui ont la même relation *inside-last* avec la même méthode *GuiElement.informObservers*, ces deux relations sont donc dites non uniformes. Le reste, excepté les relations vides, ce sont des relations d'exécution uniformes. L'étape suivante consiste à déterminer les relations d'exécution transverses dans l'ensemble des relations d'exécution uniformes non vides :

```

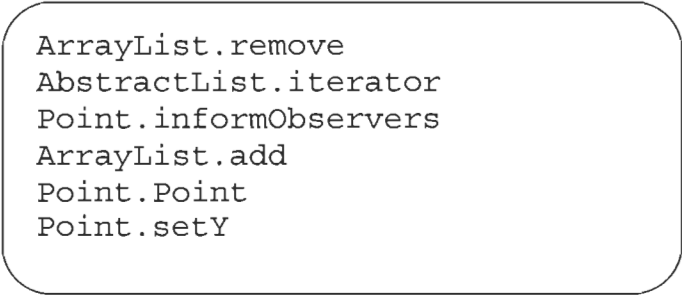
inside-first :
AbstractList.iterator#GuiElement.informObservers
ArrayList.add#Point.attachObservers
ArrayList.remove#Point.dataachObservers
Point.informObservers#Point.setXY
AbstractList.iterator#Point.informObservers
Point.informObservers#Point.setX
ArrayList.remove#GuiElement.dataachObservers
GuiElement.informObservers#GuiElement.setColor
Point.informObservers#Point.setY
ArrayList.add#GuiElement.attachObservers
Inside-last :
ArrayList.add@Point.attachObservers
ArrayList.remove@Point.dataachObservers
Point.informObservers@Point.setXY
Point.informObservers@Point.setX
ArrayList.remove@GuiElement.dataachObservers
GuiElement.informObservers@GuiElement.setColor
Point.informObservers@Point.setY
ArrayList.add@GuiElement.attachObservers
Outside-before :
AbstractList.iterator>Iterator.hasNext
Observer.update>Iterator.next
Point.Point>Point.setX
Point.setY>Point.Point
Point.Point>Point.setXY
Outside-after :
Iterator.hasNext<AbstractList.iterator
Iterator.next<Observer.update
Point.setY<Point.setX
Point.Point<Point.setY
Point.setY<Point.setXY

```

Figure 19 Les relations d'exécutions uniformes et transverses.

Toute méthode qui a la même relation d'exécution avec plus d'une seule méthode présente un aspect candidat et la relation avec ces méthodes est dite transverse, comme la méthode *Point.Point* (constructeur) au niveau des relations *outside-before*. Elle est exécutée une fois avant *Point.setX* et une autre fois avant *Point.setXY*. Les relations d'exécution soulignées sont des relations uniformes non vides et transverses.

Parmi les méthodes détectées comme aspect candidat, la méthode *Point.informObserver()*. Celle-ci est exécutée d'une façon récurrente dans des endroits différents (*setX()*, *setY()*, *setXY()*). Elle est aussi étiquetée dans la figure 14 par B, car elle représente une fonction secondaire entremêlée avec la fonction principale des trois méthodes *setX()*, *setY()* et *setXY()*. Nous observons aussi que nous avons détecté les trois méthodes dispersées *ArrayList.add*, *ArrayList.remove* et *AbstractList.iterator* étiquetées dans la figure 14 par C. D'après le résultat obtenu, les méthodes qui sont considérées comme aspects candidats sont :



```
ArrayList.remove  
AbstractList.iterator  
Point.informObservers  
ArrayList.add  
Point.Point  
Point.setY
```

Figure 20 Les aspects candidats détectés par la technique 1 de l'exemple 1.

5.2.3 *Évaluation de la technique d'analyse formelle de concepts basée sur les scénarios*

Comme nous avons discuté dans le chapitre quatre, cette technique est organisée en quatre étapes :

- ✓ Génération des chemins d'exécutions
- ✓ Définition du contexte formel ainsi que la liste des concepts
- ✓ Présentation graphique du treillis de concepts
- ✓ Identification des préoccupations transverses.

Nous utilisons la même liste des chemins d'exécution de l'exemple en cours pour chercher les scénarios ainsi que la liste des appels de méthodes qui vont être présentées comme attributs dans le contexte formel.

Voici les quatre scénarios définis à partir des chemins d'exécution :

O_1 : Point.Point, Point.setX, Point.informObservers, AbstractList.iterator, Iterator.hasNext, GuiElement.informObservers, AbstractList.iterator, Iterator.hasNext

O_2 : Point.Point, Point.setX, Point.informObservers, AbstractList.iterator, Iterator.hasNext, GuiElement.informObservers, AbstractList.iterator, Iterator.hasNext, Observer.update, Iterator.next

O_3 : Point.Point, Point.setX, Point.informObservers, AbstractList.iterator, Iterator.hasNext, Observer.update, Iterator.next, GuiElement.informObservers, AbstractList.iterator, Iterator.hasNext

O_4 : Point.Point, Point.setX, Point.informObservers, AbstractList.iterator, Iterator.hasNext, Observer.update, Iterator.next, GuiElement.informObservers, AbstractList.iterator, Iterator.hasNext, Observer.update, Iterator.next

Figure 21 L'ensemble des scénarios de l'exemple 1.

Pour simplifier la présentation du contexte formel, nous abrégeons les noms des méthodes de cette façon :

Tableau 9 Abréviation des noms des méthodes.

NOMS ABRÉGÉS	NOMS COMPLETS
P.P	POINT.POINT
P.SX	POINT.SETX
P.IO	POINT.INFORMOBSERVERS
AL.I	ABSTRACTLIST.ITERATOR
I.HN	ITERATOR.HASNEXT
GE.IO	GUIELEMENT.INFORMOBSERVERS
O.UD	OBSERVER.UPDATE
I.N	ITERATOR.NEXT

Nous présentons ensuite la relation entre les quatre scénarios (objets) avec leurs appels de méthodes de la façon suivante :

Tableau 10 Contexte formel de l'exemple 1.

	P.P	P.SX	P.IO	AL.I	I.HN	GE.IO	O.UD	I.N
O ₁	X	X	X	X	X	X		
O ₂	X	X	X	X	X	X	X	X
O ₃	X	X	X	X	X	X	X	X
O ₄	X	X	X	X	X	X	X	X

L'étape suivante consiste à définir les concepts à partir du contexte formel ci-dessus :

$$\begin{aligned}
 \text{bot} &= (\{O_2, O_3, O_4\} \{ P.P, P.SX, P.IO, AL.I, I.HN, GE.IO, O.UD, I.N \}) \\
 C_1 &= (\{O_1\} \{ P.P, P.SX, P.IO, AL.I, I.HN, GE.IO \}) \\
 C_2 &= (\{O_2\} \{ P.P, P.SX, P.IO, AL.I, I.HN, GE.IO, O.UD, I.N \}) \\
 C_3 &= (\{O_3\} \{ P.P, P.SX, P.IO, AL.I, I.HN, GE.IO, O.UD, I.N \}) \\
 C_4 &= (\{O_4\} \{ P.P, P.SX, P.IO, AL.I, I.HN, GE.IO, O.UD, I.N \}) \\
 \text{top} &= (\{O_1, O_2, O_3, O_4\} \{ P.P, P.SX, P.IO, AL.I, I.HN, GE.IO \})
 \end{aligned}$$

Figure 22 Les différents concepts de l'exemple 1.

D'après la liste des concepts, nous remarquons que :

L'extension de C_2 , C_3 et C_4 appartient à l'extension de *bot* et l'intension de C_2 , C_3 et C_4 appartient à l'intension de *bot*, d'où nous pourrions négliger les concepts C_2 , C_3 et C_4 et garder *bot* dans le treillis de concepts. La même chose pour le concept C_1 . Son extension appartient à l'extension du concept *top* et son intension appartient à l'intension du concept *top*. Donc, nous éliminons le C_1 et nous conservons le *top*. Donc, la liste des concepts ainsi que le treillis de concepts de cet exemple sont :

$$\begin{aligned} \text{bot} &= (\{O_2, O_3, O_4\} \{ \text{P.P, P.SX, P.IO, AL.I, I.HN, GE.IO, O.UD, I.N} \}) \\ \text{top} &= (\{O_1, O_2, O_3, O_4\} \{ \text{P.P, P.SX, P.IO, AL.I, I.HN, GE.IO} \}) \end{aligned}$$

Figure 23 La liste finale des concepts de l'exemple 1.

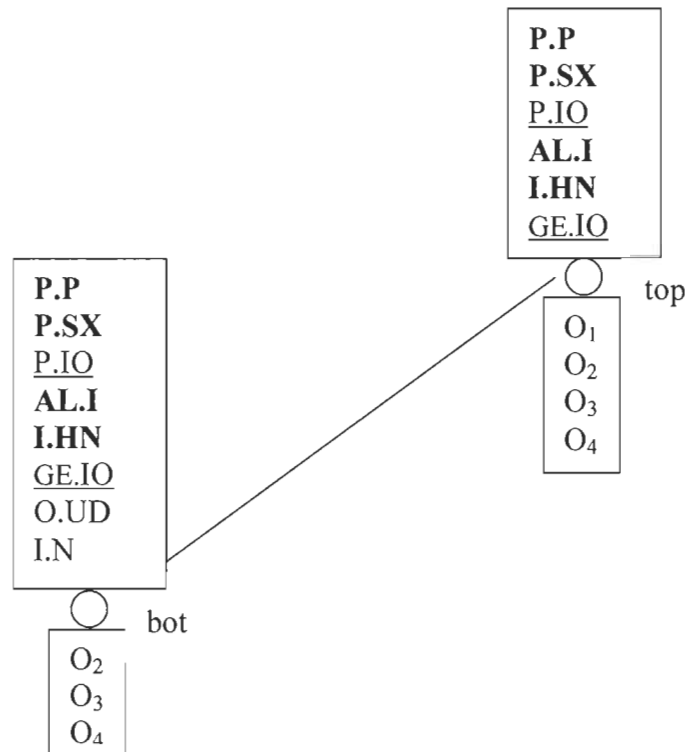


Figure 24 Treillis de concepts de l'exemple 1.

Le treillis de concept présenté dans la figure 24 montre clairement les méthodes qui vérifient les deux contraintes discutées dans le chapitre quatre.

Contrainte 1 :

Les deux concepts top et bot comportent la méthode IO (InformObservers) soulignée qui appartient aux deux classes Point et GuiElement.

Contrainte 2 :

Plusieurs méthodes de la même classe comme point, setX, informObservers, iterator, hasNext (méthodes en gras) contribuent aux deux concepts top et bot. D'où, les deux concepts présentent des préoccupations transverses ainsi les méthodes suivantes sont des aspects candidats :

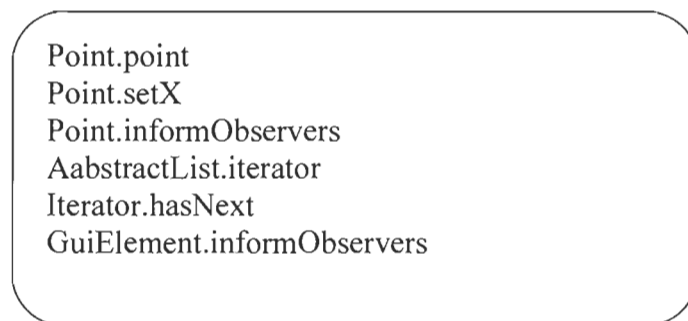


Figure 25 Les aspects candidats détectés par la technique 2 de l'exemple 1.

Nous observons qu'il y'a trois méthodes *Point.point*, *Point.informObservers* et *AabstractList.iterator* détectées par les deux techniques pour leurs appels récurrents et à cause du fait qu'elles appartiennent à plus d'un scénario. La méthode *Point.informObservers* (étiqueté B dans la figure 14) représente le code relatif au mécanisme de notification de l'observateur entremêlée avec la fonctionnalité principale ainsi que la méthode *AabstractList.iterator* est une méthode dispersée (étiqueté C dans la

figure 14). Il y a aussi des méthodes qui sont détectées par l'une des deux techniques soit à cause de leurs appels récurrents ou soit elles vérifient l'une des deux contraintes de la technique FCA.

5.3 Exemple 2 (la classe *TangledStack*)

5.3.1 Présentation de l'exemple

Cet exemple est décrit dans [21]. Il présente une classe nommée *Tangledstack* qui elle-même contient des méthodes de gestion d'une pile. Cette classe assure deux fonctionnalités :

Fonctionnalité principale : Empiler, dépiler, vérifier si la pile est pleine ou vide.

Fonctionnalité secondaire : Affichage du contenu de la pile dans un champ de texte d'une fenêtre (*frame*). Cette dernière était dotée d'une seconde responsabilité (bout de code étiqueté A), cela a rendu la classe *Tangledstack* enchevêtrée et ambiguë.

5.3.2 Évaluation de la technique d'analyse des appels récurrents et transverses basée sur les scénarios

Le code source de l'exemple 2 :

```

import javax.swing.*;
public class Tangledstack {
    private int _top = -1;
    private Object[] _elements;
    private final int S_SIZE = 10;
    private JLabel _Label = new JLabel("stack");
    private JTextField _text = new JTextField(20);
    public Tangledstack(JFrame frame) {
        elements = new Object[S_SIZE];
        frame.getContentPane().add(_Label);
        _text.setText("[]");
        frame.getContentPane().add(_text);
    }
    public String toString() {
        StringBuffer result = new StringBuffer("[");
        for(int i = 0; i<= _top; i++){
            result.append(_elements[i].toString());
            if(i != _top)
                result.append(", ");
        }
        result.append("]");
        return result.toString();
    }
    private void display() {
        _text.setText(toString());
    }
    public void push(Object element) {
        elements[++ _top] = element;
        display();
    }
    public void pop() {
        _top--;
        display();
    }
    public Object top() {
        return _elements[_top];
    }
    public boolean isFull(){
        return (_top == S_SIZE - 1);
    }
    public boolean isEmpty(){
        return (_top < 0);
    }
}

```

Figure 26 Code de la classe TangledStack [21].

Génération des chemins d'exécution

Nous citons toutes les méthodes déclarées dans la classe *Tangledstack* avec leurs appels de méthodes :

```
Tangledstack.push:Tangledstack.display
Tangledstack.Tangledstack:Container.add,JFrame.getContentPane,JTextCom
ponent.setText,Container.add,JFrame.getContentPane
Tangledstack.isEmpty:
Tangledstack.display:JTextComponent.setText,Tangledstack.toString
Tangledstack.pop:Tangledstack.display
Tangledstack.toString:StringBuffer.StringBuffer,{StringBuffer.append,Object
.toString,(StringBuffer.append)},StringBuffer.append,StringBuffer.toString
Tangledstack.top:
Tangledstack.isFull:
```

Figure 27 L'ensemble des chemins d'exécution de l'exemple 2.

À partir de cet ensemble de chemins d'exécution nous extrairons l'ensemble des relations d'exécution. Voici ce que nous obtenons:

Extraction des quatre relations d'exécution :

Inside first (symbole #) :

Container.add # Tangledstack.Tangledstack

JTextComponent.setText # Tangledstack.display

Tangledstack.display # Tangledstack.push

Tangledstack.display # Tangledstack.pop

StringBuffer.StringBuffer # Tangledstack.toString

Inside last (symbole @):

JFrame.getContentPane @ Tangledstack.Tangledstack

Tangledstack.toString @ Tangledstack.display

Tangledstack.display @ Tangledstack.pop

StringBuffer.toString @ Tangledstack.toString

Tangledstack.display @ Tangledstack.push

Outside before (symbole >):

Container.add > JFrame.getContentPane

JFrame.getContentPane > JTextComponent.setText

JTextComponent.setText > Container.add

JTextComponent.setText > Tangledstack.toString

StringBuffer.StringBuffer > StringBuffer.append

StringBuffer.append > Object.toString

Object.toString > StringBuffer.append

StringBuffer.append > StringBuffer.append

StringBuffer.append > StringBuffer.toString

Figure 28 Les relations d'exécutions de l'exemple 2.

L'ensemble des relations d'exécution ci-dessus contient des relations non uniformes, comme les deux premières relations soulignées dans *inside-first* où il y a deux méthodes différentes *Container.add* et *JTextComponent.setText* qui ont la même relation avec la

méthode *Tangledstack.Tangledstack*. Toutes les relations trouvées non uniformes sont soulignées.

Identification des relations d'exécution transverses :

Inside first (#):

Tangledstack.display # *Tangledstack.push*

Tangledstack.display # *Tangledstack.pop*

Inside last (@):

Tangledstack.display @ *Tangledstack.pop*

Tangledstack.display @ *Tangledstack.push*

Outside before (>):

JTextComponent.setText > *Container.add*

JTextComponent.setText > *Tangledstack.toString*

StringBuffer.append > *Object.toString*

StringBuffer.append > *StringBuffer.append*

StringBuffer.append > *StringBuffer.toString*

Outside after (<)

StringBuffer.append < *Object.toString*

StringBuffer.append < *StringBuffer.append*

Figure 29 Les relations d'exécutions transverses de l'exemple 2.

Donc, la liste des aspects candidats est : *Tangledstack.display*, *JTextComponent.setText*, *StringBuffer.append*. Nous remarquons que nous avons détecté deux méthodes parmi les méthodes de la fonctionnalité secondaire étiquetée par A dans la figure 26 précédente (*Tangledstack.display* et *JTextComponent.setText*).

5.3.3 Évaluation de la technique d'analyse formelle de concepts basée sur les scénarios

Avant de tracer le contexte formel de cet exemple, nous abrégeons les noms des méthodes de la façon suivante :

Tableau 11 Abréviation de noms des méthodes.

Noms abrégés	Noms complets
TS.push	Tangledstack.push
TS.disp	Tangledstack.display
C.add	Container.add
JF.gcp	JFrame.getContentPane
JT.st	JTextComponent.setText
TS.ie	Tangledstack.isEmpty
TS.str	Tangledstack.toString
TS.pop	Tangledstack.pop
SB.SB	StringBuffer.StringBuffer
SB.app	StringBuffer.append
O.str	Object.toString
SB.str	StringBuffer.toString
TS.top	Tangledstack.top
TS.if	Tangledstack.isFull

Le contexte formel suivant décrit la relation entre l'ensemble des scénarios de cet exemple avec les appels de méthodes :

Tableau 12 Le contexte formel de l'exemple 2.

	TS.DISPC.ADD	F.GCP	T.ST	TS.STR	SB.SB	SB.APP	O.STR	SB.STR
O ₁	X							
O ₂		X	X					
O ₃			X	X				
O ₄	X							
O ₅					X	X	X	X

À partir du contexte formel précédent, nous définissons la liste des concepts de cet exemple :

$$\begin{aligned} \text{Bot} &= (\{\} \{ \text{TS.disp}, \text{C.add}, \text{JF.gcp}, \text{JT.st}, \text{TS.str}, \text{SB.SB}, \text{SB.app}, \\ &\quad \text{O.str}, \text{SB.str} \}) \\ \text{C}_1 &= (\{ \text{O}_1, \text{O}_4 \} \{ \text{TS.disp} \}) \\ \text{C}_2 &= (\{ \text{O}_2 \} \{ \text{C.add}, \text{JF.gcp}, \text{JT.st} \}) \\ \text{C}_3 &= (\{ \text{O}_3 \} \{ \text{JT.st}, \text{TS.str} \}) \\ \text{C}_4 &= (\{ \text{O}_2, \text{O}_3 \} \{ \text{JT.st} \}) \\ \text{C}_5 &= (\{ \text{O}_5 \} \{ \text{SB.sb}, \text{SB.app}, \text{O.str}, \text{SB.str} \}) \\ \text{Top} &= (\{ \text{O}_1, \text{O}_2, \text{O}_3, \text{O}_4, \text{O}_5 \} \{ \}) \end{aligned}$$

Figure 30 La liste des concepts de l'exemple 2.

Ensuite, nous présentons ces concepts sur un graphe pour que nous dévoilions clairement les aspects candidats.

Le treillis de concepts :

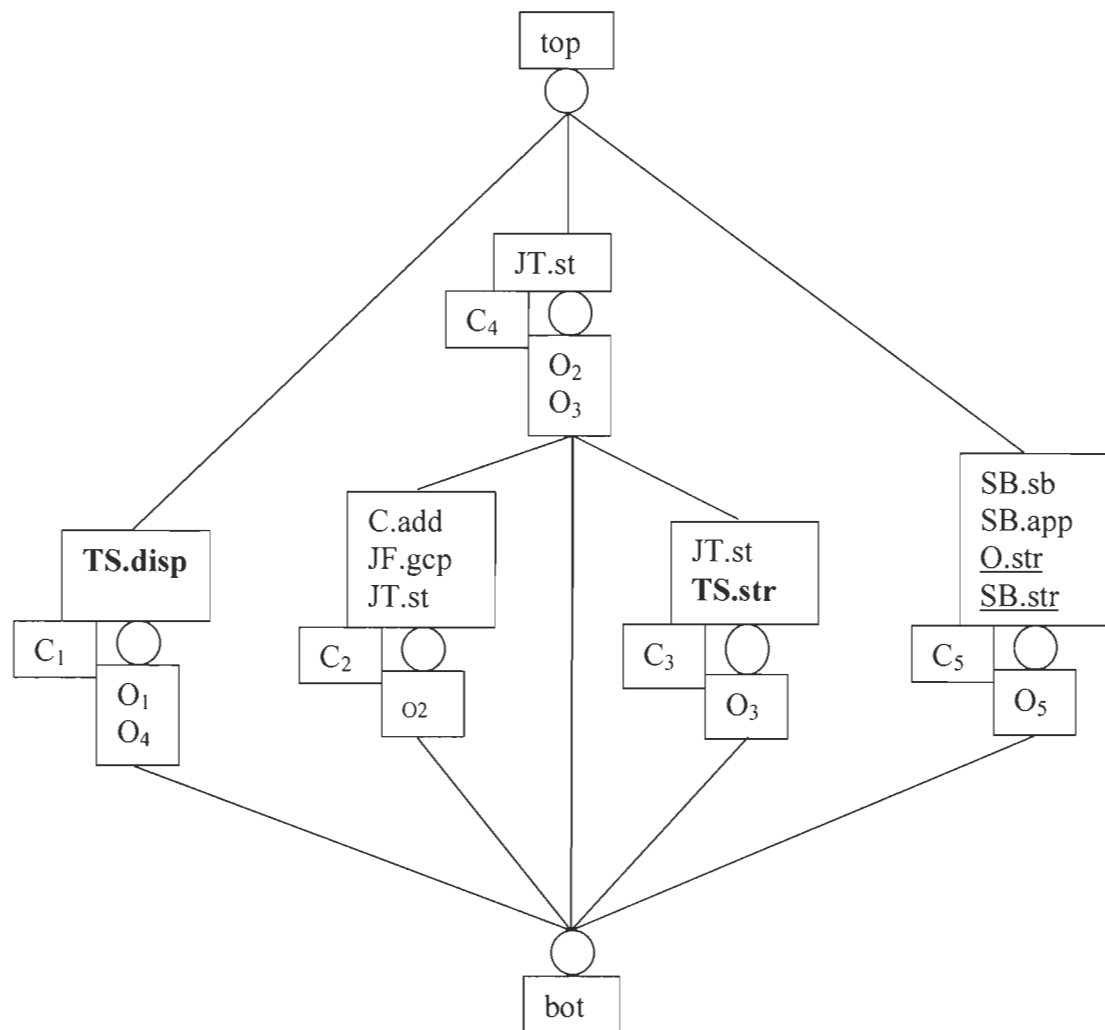


Figure 31 Le treillis de concepts de l'exemple 2.

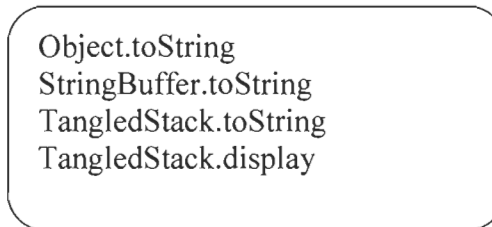
Les préoccupations transverses :

Contrainte 1 :

Nous pouvons déduire d'après le treillis de concepts obtenu que le concept C_5 contient la méthode `str` (`toString`) soulignée qui appartient aux deux classes `O` (`Object`) et `SB` (`StringBuffer`).

Contrainte 2 :

Les deux méthodes en gras *str* (*toString*) et *disp* (*display*) appartiennent à la même classe TS (*TangledStack*) et contribuent aux deux scénarios O_1 et O_3 . D'où, la liste des méthodes qui présentent des aspects candidats détectés par cette technique sont :



```
Object.toString
StringBuffer.toString
TangledStack.toString
TangledStack.display
```

Figure 32 Les aspects candidats détectés par la technique 2 de l'exemple 2.

À première vue, la technique FCA n'a pas détecté toutes les méthodes considérées comme des aspects candidats comme *JTextComponent.setText*, la seule méthode détectée appartenant à la fonctionnalité secondaire est la méthode *TangledStack.display*.

TangledStack.display est la seule méthode détectée par les deux techniques, car elle est appelée dans deux méthodes différentes *push* et *pop* de la classe *TangledStack* et elle contribue à deux scénarios différents. Nous observons aussi que la méthode *TangledStack.display* appartient aux méthodes de la fonctionnalité secondaire, pour le reste des méthodes les deux techniques ont détecté des méthodes non candidates comme *StringBuffer* pour la technique des appels récurrents et *StringBuffer.toString* pour la technique FCA.

5.4 Exemple 3 (Design pattern chaîne de responsabilité)

5.4.1 Présentation de l'exemple

Idéalement, chaque classe devrait jouer un rôle unique mais souvent cela n'est pas possible. La double responsabilité peut être trouvée dans des classes qui jouent des rôles multiples. Comme exemple, nous considérons une chaîne de responsabilités. Son but est de permettre à une chaîne d'objets de traiter une demande, sans qu'aucun d'eux ne connaisse les capacités des autres objets, qui peuvent être de différents types. Dès qu'un objet reçoit une demande qu'il ne peut pas gérer, il passe à l'objet suivant dans la chaîne jusqu'à ce que la demande arrive à l'objet concerné.

La figure 33 montre une implémentation possible du patron Chaîne de responsabilités par une classe *ColorImage* [21]. Le rôle secondaire est modélisé par l'interface *Chain*, que tous les objets participants doivent implémenter (le code relatif à ce rôle est ombré, désigné par une étiquette). En bref, la classe *ColorImage* implémente l'interface *chain* pour déclarer les méthodes suivantes :

- *addChain()* : ajoute une autre classe à la chaîne des classes.
- *getChain()* : retourne l'objet courant auquel les messages sont adressés.
- *sendToChain()* : achemine un message à l'objet suivant dans la chaîne.

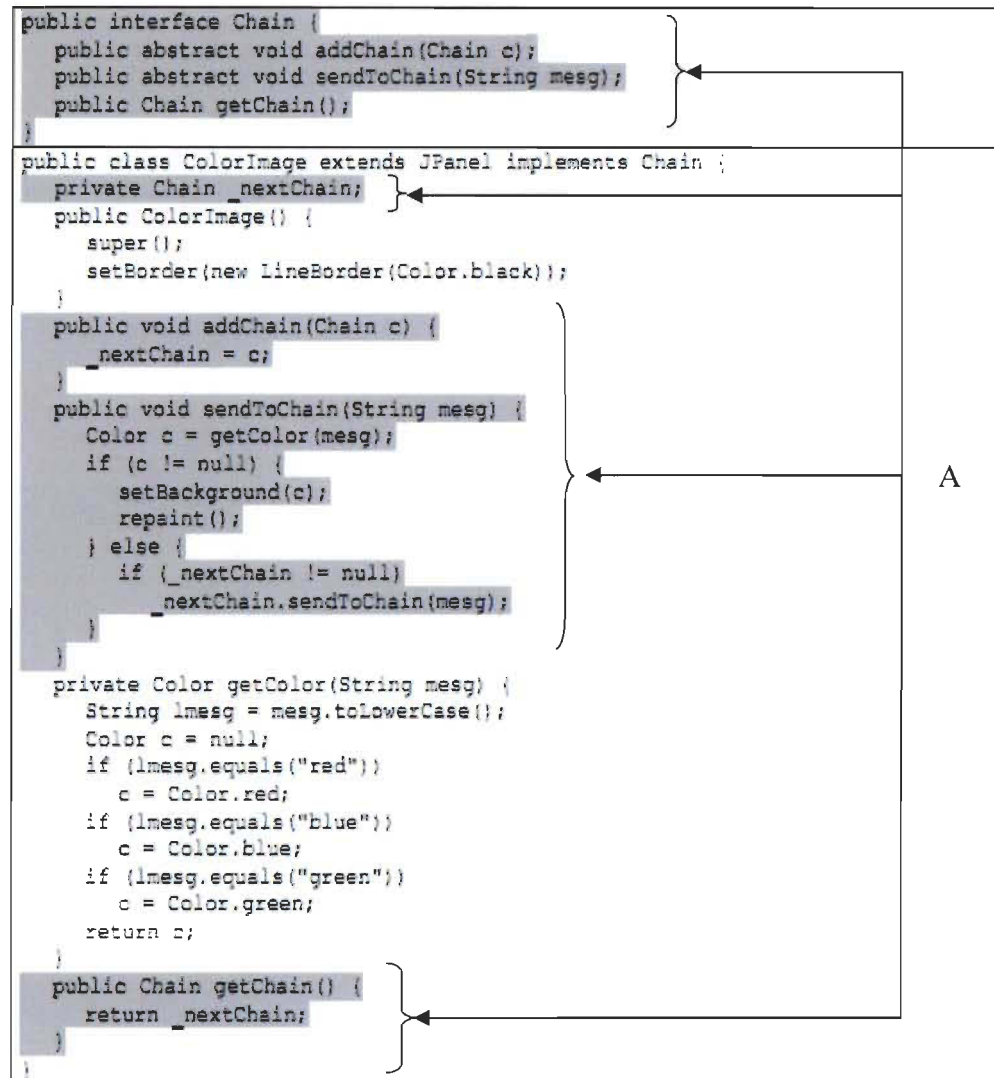


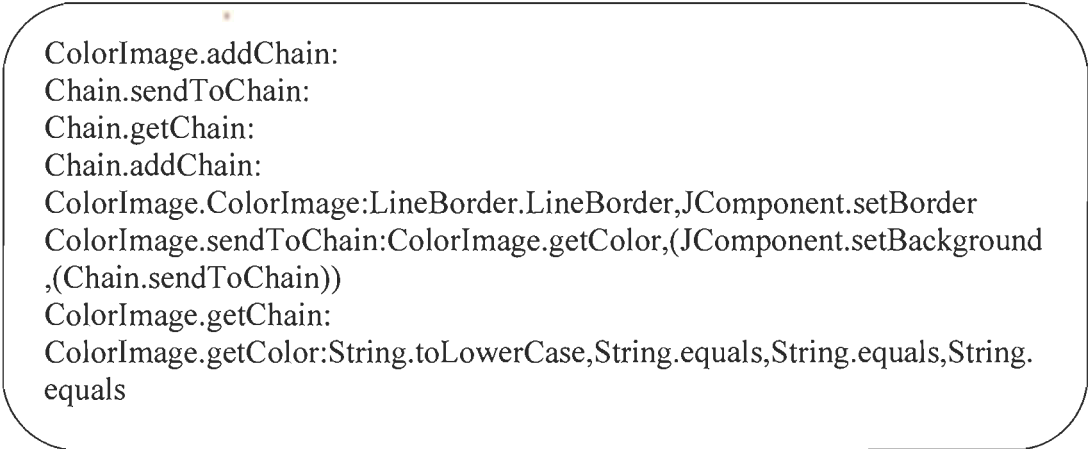
Figure 33 Exemple d'implémentation du patron chaîne de responsabilités (la classe *ColorImage*) [21].

En fait, nous voyons bien que la classe a été dotée d'une seconde responsabilité relative à la manipulation des messages reçus et leur passation aux objets suivants dans la chaîne d'objets participants au cas où l'objet en cours ne pourrait pas saisir le message.

5.4.2 *Évaluation de la technique d'analyse des appels récurrents et transverses basée sur les scénarios*

Nous commençons par extraire les chemins d'exécution, voici la liste des chemins d'exécution :

Génération manuelle des quatre relations d'exécution



```

ColorImage.addChain:
Chain.sendToChain:
Chain.getChain:
Chain.addChain:
ColorImage.ColorImage:LineBorder.LineBorder,JComponent.setBorder
ColorImage.sendToChain:ColorImage.getColor,(JComponent.setBackground
,(Chain.sendToChain))
ColorImage.getChain:
ColorImage.getColor:String.toLowerCase,String.equals,String.equals,String.
equals

```

Figure 34 La liste des chemins d'exécution de l'exemple 3.

L'étape suivante consiste à définir les quatre types de relations d'exécution :

Définition des relations d'exécution

Inside first (symbole #) :

LineBorder.LineBorder#ColorImage.ColorImage

ColorImage.getColor#ColorImage.sendToChain

String.toLowerCase#ColorImage.getColor

Inside last (symbole @):

JComponent.setBorder@ColorImage.ColorImage

JComponent.setBackground@ColorImage.sendToChain

Chain.sendToChain@ColorImage.sendToChain

String.equals@ColorImage.getColor

Outside before (symbole >):

Chain.sendToChain>Chain.sendToChain

LineBorder.LineBorder>JComponent.setBorder

ColorImage.getColor>JComponent.setBackground

String.toLowerCase>String.equals

String.equals>String.equals

ColorImage.getColor>Chain.sendToChain

Outside after (symbole <):

Chain.sendToChain<Chain.sendToChain

JComponent.setBorder<LineBorder.LineBorder

JComponent.setBackground<ColorImage.getColor

String.equals<String.toLowerCase

String.equals<String.equals

Chain.sendToChain<ColorImage.getColor

Figure 35 Les relations d'exécution de l'exemple 3.

Nous procédons de la même façon que pour les exemples précédents. Nous cherchons toutes les méthodes différentes qui ont la même relation avec la même méthode et nous considérons leurs relations comme étant des relations non uniformes.

Après élimination des relations d'exécution non uniformes, nous identifions les préoccupations transverses.

Identification des relations d'exécution uniformes :

Inside first (symbole #) :

LineBorder.LineBorder#ColorImage.ColorImage
 ColorImage.getColor#ColorImage.sendToChain
 String.toLowerCase#ColorImage.getColor

Inside last (symbole @):

JComponent.setBorder@ColorImage.ColorImage
 String.equals@ColorImage.getColor

Outside before (symbole >):

LineBorder.LineBorder>JComponent.setBorder
 ColorImage.getColor>JComponent.setBackground

Outside after (symbole <):

Chain.sendToChain<Chain.sendToChain
 JComponent.setBorder<LineBorder.LineBorder
String.equals<String.toLowerCase
String.equals<String.equals

Figure 36 Les relations d'exécution uniformes et transverses de l'exemple 3.

La seule méthode qui a une relation avec deux méthodes différentes est la méthode *String.equals*. La technique n'a pas détecté le code étiqueté par A, cela explique que cette technique détecte les appels récurrents et transverses et non pas l'enchevêtrement dans le code.

5.4.1 Évaluation de la technique d'analyse formelle de concepts basée sur les scénarios

Nous suivrons les mêmes étapes des deux exemples précédentes. Nous commençons tout d'abord par donner des noms abrégés aux méthodes.

Tableau 13 Abréviation des noms des méthodes.

Noms abrégés	Noms complets
CI.ac	ColorImage.addChain
C.stc	Chain.sendToChain
C.gch	Chain.getChain
C.ac	Chain.addChain
CI.ci	ColorImage.ColorImage
LB.lb	LineBorder.LineBorder
JC.sb	JComponent.setBorder
CI.stc	ColorImage.sendToChain
CI.gco	ColorImage.getColor
JC.sbg	JComponent.setBackground
CI.gch	ColorImage.getChain
S.tlc	String.toLowerCase
S.e	String.equals

Dans le tableau suivant nous présentons les différents scénarios de l'exemple 3 ainsi que leurs appels de méthodes :

Tableau 14 Le contexte formel de l'exemple 3.

	ci.ac	c.stc	c.gch	c.ac	ci.ci	b.lb	c.sb	ci.stc	ci.gco	c.sbg	ci.gch	s.tlc	s.e
O ₁	X												
O ₂		X											
O ₃			X										
O ₄				X									
O ₅					X	X	X						
O ₆		X						X	X	X			
O ₇											X		
O ₈									X			X	X

Ensuite, nous définissons l'ensemble des concepts extraits du contexte formel précédent afin de présenter le treillis de concepts et identifier les aspects candidats possibles.

La liste des concepts :

$\text{bot} = (\{\}, \{\text{Cl.ac}, \text{c.stc}, \text{c.gch}, \text{c.ac}, \text{ci.ci}, \text{lb.lb}, \text{jc.sb}, \text{ci.stc}, \text{ci.gco}, \text{jc.sbg}, \text{ci.gch}, \text{s.tlc}, \text{S.e}\})$
 $C_1 = (\{O_1\}, \{\text{Cl.ac}\})$
 $C_3 = (\{O_3\}, \{\text{C.gch}\})$
 $C_4 = (\{O_4\}, \{\text{C.ac}\})$
 $C_5 = (\{O_5\}, \{\text{Cl.ci}, \text{LB.lb}, \text{JC.sb}\})$
 $C_6 = (\{O_6\}, \{\text{C.stc}, \text{Cl.stc}, \text{Cl.gco}, \text{JC.sbg}\})$
 $C_7 = (\{O_7\}, \{\text{Cl.gch}\})$
 $C_8 = (\{O_8\}, \{\text{Cl.gco}, \text{S.tlc}, \text{S.e}\})$
 $C_9 = (\{O_2, O_6\}, \{\text{C.stc}\})$
 $C_{10} = (\{O_6, O_8\}, \{\text{Cl.gco}\})$
 $\text{top} = (\{O_1..O_8\}, \{\})$

Figure 37 La liste des concepts de l'exemple 3.

Voici le treillis de concepts correspond :

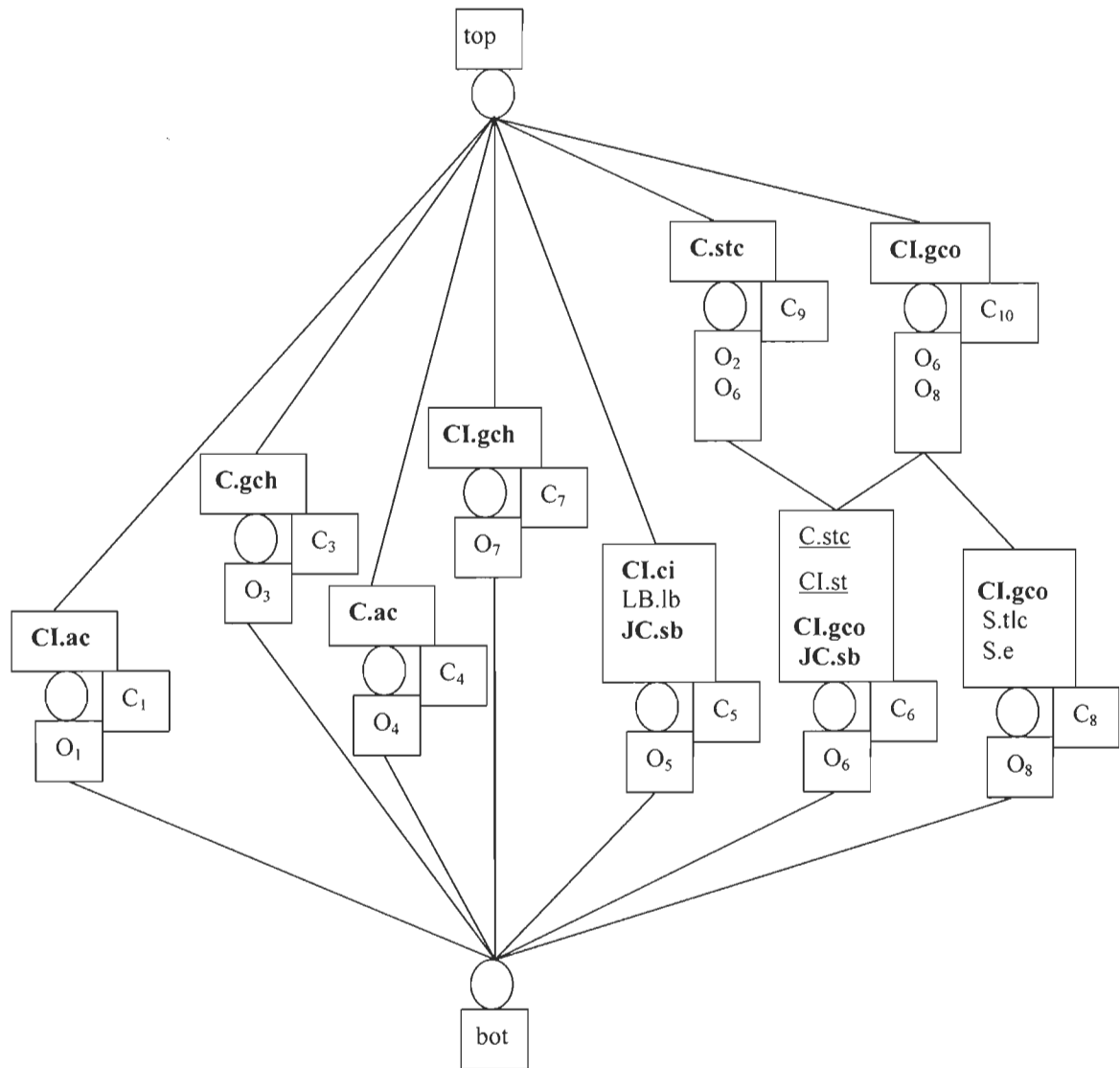


Figure 38 Le treillis de concepts de l'exemple 3.

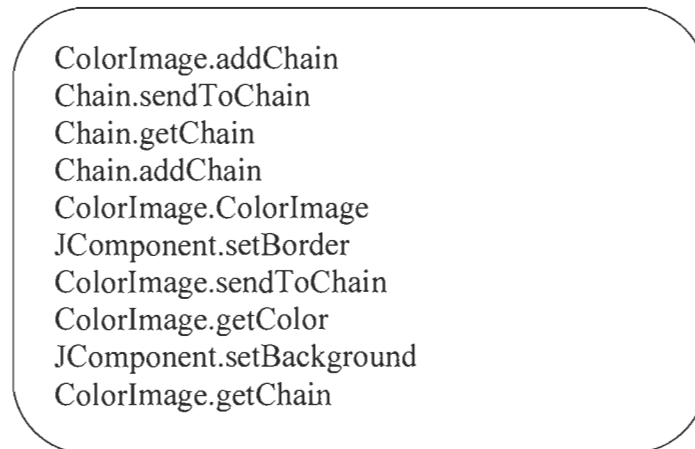
Contrainte 1 :

Nous observons que la méthode *stc* (*sendToChain*) appartient à deux classes différentes *Chain* et *ColorImage*.

Contrainte 2 :

Les méthodes en gras sont des méthodes différentes qui appartiennent à la même classe mais contribuent à des concepts différents. Comme les méthodes *sb* (*setBorder*) et *sbg* (*setBackground*) de la classe *JComponent* contribuent à deux scénarios O_5 et O_6

Voici l'ensemble des aspects candidats détectés par la technique :



```

ColorImage.addChain
Chain.sendToChain
Chain.getChain
Chain.addChain
ColorImage.ColorImage
JComponent.setBorder
ColorImage.sendToChain
ColorImage.getColor
JComponent.setBackground
ColorImage.getChain
  
```

Figure 39 Les aspects candidats de l'exemple 3 détectés par la technique 2.

Contrairement à la technique précédente, cette technique a détecté toutes les méthodes ombrées dans la figure 33 ci-dessus (considérée comme fonctionnalité secondaire) comme *getChain*, *sendToChain*. Cependant, elle a détecté aussi des méthodes qui ne sont pas considérées comme aspects candidats comme le constructeur *ColorImage* et la méthode *getColor*, car ce sont des méthodes qui assurent la fonctionnalité principale de la classe *ColorImage*. Elles sont détectées par la technique puisque elles vérifient l'une des deux contraintes discutées précédemment.

5.5 Etude de cas

Avec le nombre important et la variété des techniques d'*aspects mining* proposées dans la littérature, il devient de plus en plus intéressant de comparer d'une façon méthodologique ces techniques afin d'évaluer leur cohérence, leur compatibilité et leurs résultats [22]. Ce chapitre revient sur les deux techniques discutées précédemment en proposant une évaluation empirique basée sur un *Framework* connu *JHotDraw*. Ce système est largement utilisé dans la littérature. C'est un programme java composé de 311 classes et 2135 méthodes sans compter les classes et les méthodes de test. *JHotDraw* est un *framework* de dessin de graphes 2D. Ce projet était à l'origine développé comme un exercice pour illustrer un bon usage de la conception orientée objet. Nous utilisons, en fait, les deux versions de ce programme : la version originale Java et celle en AspectJ (résultat du refactoring aspect du programme original). Nous supposons, toutefois, que l'étude de cas a été bien conçue et que les précautions ont été prises pour séparer proprement les préoccupations transverses et les rendre aussi compréhensibles que possible. Néanmoins, *JHotDraw* expose quelques limitations de modularisation (trouvées même dans des systèmes bien conçus), et contient des préoccupations transverses intéressantes [23].

L'évaluation a également intégré, pour des fins de comparaison, les résultats fournis par une troisième technique statique (*Fan-in*) décrite dans la section 2.4.5 du chapitre 2, connue dans la littérature du domaine.

Le but de cette évaluation comparative n'est pas d'identifier la meilleure technique, mais plutôt de comparer les techniques et évaluer, leurs principales forces et faiblesses. En outre, en déterminant par exemple où les techniques se chevauchent et où elles sont

complémentaires, cela nous permettra peut être d'identifier et de proposer des (pistes de) combinaisons intéressantes et d'appliquer ces combinaisons sur le même « *benchmark* » pour vérifier si nous aurons effectivement de meilleurs résultats.

Les résultats sont comparés avec la liste des classes réellement aspectualisées dans le projet *AJHotDraw-v0.4*, version aspectualisée de la version orientée objet du projet *JHotDraw60b1*, un framework java relativement important et bien conçu. *AJHotDraw-v0.4* est écrit en utilisant *AspectJ*, une extension orientée aspect de java. Cette extension est disponible dans les projets open-source *Eclipse* [24].

Les points discutés dans la suite de ce chapitre peuvent être résumés comme suit :

- ✓ Présentation des résultats de chaque technique (précisions, rappels, corrélation de spearman, corrélation de pearson et la courbe ROC),
- ✓ Combinaison des résultats de la technique (intersection des résultats)
- ✓ Combinaison des techniques (deux à deux puis les trois en même temps) et l'analyse statistique de différentes combinaisons,
- ✓ Discussion et vérification si les techniques donnent de bons résultats en les appliquant séparément ou de façon combinée.

5.5.1 Terminologies

Méthodes réellement aspectualisées : est l'ensemble des méthodes aspectualisées dans le projet *AJHotDraw-v0.4*, version aspectualisée de *JHotDraw60b1*.

Classes réellement aspectualisées : est l'ensemble des classes aspectualisées dans le projet *AJHotDraw-v0.4*, version aspectualisée de *JHotDraw60b1*.

Précision : est définie comme le nombre des aspects candidats détectés vrais (méthodes ou classes) d'un projet orienté objet (comme JHotDraw) qui sont effectivement survenus sur le nombre des classes réellement aspectualisées dans la version aspectualisée du même projet.

Recall (rappel) : est le nombre des éléments (méthodes ou classes) non détectés non réellement aspectualisés sur le total des éléments confirmés faux (les éléments non réellement aspectualisés).

5.5.2 Définitions

Corrélation de Pearson : Le but de la corrélation est de savoir s'il existe une relation entre deux variables. En général, cette relation peut être mesurée par un coefficient de corrélation de *Pearson* noté « r ». Ce coefficient est toujours compris entre -1 et +1 [26].

- $r = 0$ ou voisin de 0 signifie une absence de relation.
- $r < 0$: une relation négative.
- $r > 0$: une relation positive.
- $r = 1$ ou $r = -1$: relation parfaite.

Corrélation de Spearman : Le coefficient de corrélation de *Pearson* constitue une mesure de l'intensité de liaison linéaire entre 2 variables [26]. Cependant, il existe des situations pour lesquelles le calcul de la corrélation sur les valeurs est inadapté. Si les variables sont ordinales, discrètes, ou si des outils risquent de biaiser les résultats, ou encore que les valeurs en elles-mêmes n'ont que peu d'importance, ou qu'elles ne suivent pas une loi normale, il ne nous reste qu'à calculer les corrélations des rangs. Nous n'utilisons alors pas les valeurs des observations mais plutôt leur rang [27].

Nous calculons ensuite le coefficient de corrélation de *Spearman* « r » et de la même manière que la corrélation de *Pearson*, nous analysons la valeur r pour décider s'il existe une relation entre les rangs ou non. Dans notre cas, nous allons aussi calculer la corrélation de *Spearman* pour chaque technique ainsi que pour chaque combinaison de résultats.

La courbe ROC : La courbe *ROC* (*Receiver Operating Characteristics*) est une représentation graphique de la relation existante entre la sensibilité et la spécificité d'un test pour toutes les valeurs seuils possibles. L'ordonnée représente la sensibilité et l'abscisse correspond à la quantité (1 - spécificité). Nous allons nous concentrer sur la courbe ROC pour visualiser la performance des techniques et les différentes combinaisons afin de les comparer. Nous désignons par sensibilité la capacité de donner un résultat positif et la spécificité correspond à la capacité de donner un résultat négatif. La courbe des points (1-spécificité, sensibilité) est la courbe *ROC* [28].

L'aire sous la courbe (*Area Under the Curve-AUC*) est un indice synthétique calculé pour les courbes *ROC*. Pour un modèle idéal, nous avons $AUC = 1$. Pour un modèle aléatoire, nous avons $AUC = 0.5$, nous considérons que le modèle est bon si AUC est supérieur à 0.7. Un modèle ayant une AUC supérieur à 0.9 est excellent [28].

XSTAT : Dans nos calculs de corrélations et la représentation graphique de la courbe *ROC*, nous allons utiliser *XLSTAT-Sim* un outil puissant d'analyse de données et de statistiques. Dans notre cas, nous utilisons ce logiciel afin de décider quelle est la technique et/ou la meilleure combinaison qui peuvent donner de meilleurs résultats.

5.5.3 Application des techniques d'aspects mining

Dans cette section, nous présentons les résultats de l'application de chaque technique sur la version 60b1 de *JHotDraw*.

5.5.3.1 La technique d'analyse des appels récurrents

5.5.3.1.1 Seuil

Nous définissons le seuil comme étant le nombre d'appels récurrents dans une classe donnée. En d'autres termes, si nous fixons un seuil égal à S , alors pour chaque méthode m faisant n appels récurrents dans une classe C donnée, si $S \leq n$ la méthode m sera alors considérée comme un aspect candidat. Ensuite, nous calculons la précision et le rappel en fonction du seuil.

5.5.3.1.2 Résultats

Tableau 15 Résultats de la technique des appels récurrents (précision et rappel) en fonction du seuil.

Seuil	Classes détectées réellement aspectualisées	Classes détectées	Rappel	Précision
43	2	2	100,00	5,88
41	2	3	99,59	5,88
23	3	4	99,59	8,82
21	3	5	99,18	8,82
20	3	6	98,77	8,82
19	3	7	98,36	8,82
18	3	8	97,95	8,82
17	3	9	97,54	8,82
16	5	13	96,72	14,71
15	6	14	96,72	17,65
10	6	15	96,31	17,65
9	7	16	96,31	20,59
8	8	20	95,08	23,53
7	8	21	94,67	23,53
6	12	29	93,03	35,29
5	12	35	90,57	35,29
4	13	51	84,43	38,24
3	14	58	81,97	41,18
2	15	76	75,00	44,12

Voici un graphe qui illustre la courbe des précisions et des rappels obtenus en fonction du seuil :

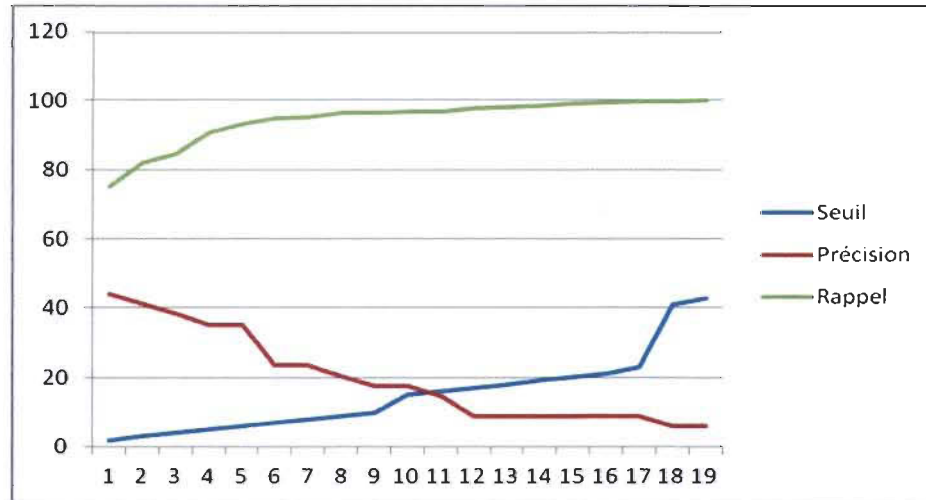


Figure 40 Précisions et rappels de la technique des appels récurrents en fonction du seuil.

5.5.3.1.3 Discussion

Nous remarquons sur le tableau ci-dessus que la meilleure précision égale à 44,12 % (inférieure à 50%) est obtenue avec un seuil égal à 2 où la technique a pu détecter 76 classes. Parmi les 76 classes détectées, il y a 15 classes réellement aspectualisées qui ont été détectées. Sachant que le nombre des classes réellement aspectualisées dans JHotDraw est égal à 34. Par contre, nous avons obtenu un rappel acceptable égal à 75% ce qui signifie que la technique avec un seuil égal à 2, le nombre des classes détectées non réellement aspectualisées n'était pas important. D'après le graphe, chaque fois que le seuil augmente, le rappel augmente, cela signifie que l'erreur diminue, mais la précision diminue aussi, ce qui explique la diminution du nombre de classes détectées réellement aspectualisées.

5.5.3.2 La technique d'analyse formelle de concepts

5.5.3.2.1 Seuil

Nous avons vu dans le chapitre quatre que pour considérer certaines méthodes comme étant des aspects candidats, il faut que les deux contraintes suivantes soient vérifiées :

- ✓ Scattering : les méthodes qui appartiennent à plus d'une classe,
- ✓ Tangling : les différentes méthodes d'une même classe qui contribuent dans plus d'un cas d'utilisation.

Pour définir un seuil, nous nous sommes basés sur la deuxième condition (*Tangling*) puisque nous avons obtenu des résultats meilleurs que les résultats obtenus en utilisant un seuil basé sur la première contrainte (*Scattering*).

En d'autres termes, le seuil est le nombre de cas d'utilisation appelant les méthodes d'une classe candidate. C'est-à-dire, si nous fixons un seuil égal à 3, alors une classe C est considérée comme candidate si l'une de ces méthodes contribue à 3 ou plus de 3 cas d'utilisation.

5.5.3.2.2 Résultats

Tableau 16 Résultats de la technique FCA (précision et rappel) en fonction du seuil.

Seuil	Classes détectées réellement aspectualisées	Classes détectées	Rappel	Précision
7	0	0	100,00	0,00
6	7	25	92,62	20,59
5	8	30	90,98	23,53
4	14	40	89,34	41,18
3	16	44	88,52	47,06
2	18	50	86,89	52,94

Voici le graphe correspondant :

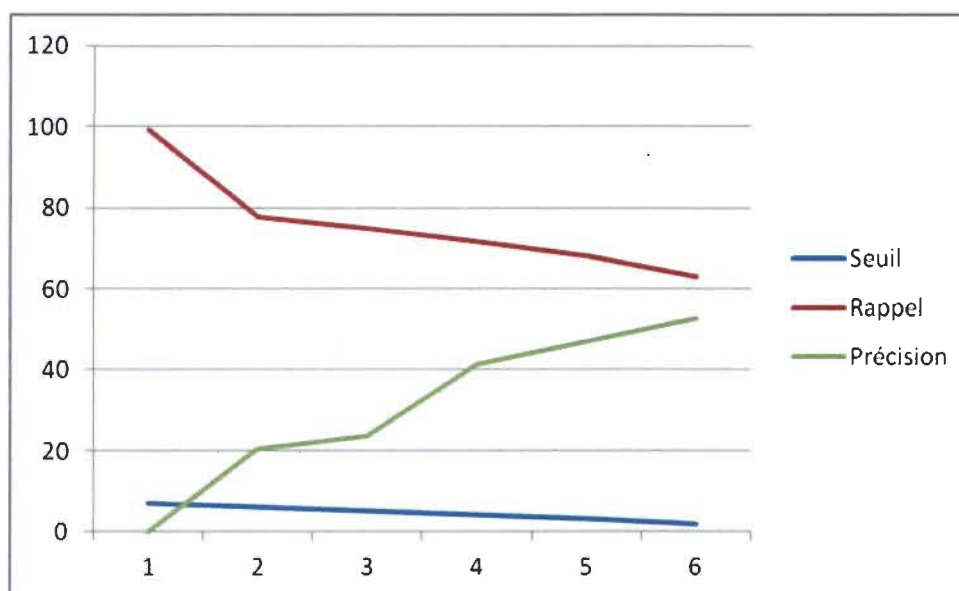


Figure 41 Précisions et rappels de la technique FCA en fonction du seuil.

5.5.3.2.3 Discussion

Le tableau nous montre que la meilleure précision est de 52.63 % au niveau d'un seuil égal à 2. Cette technique a détecté avec ce seuil 50 classes. Nous constatons aussi qu'elle a détecté 18 classes parmi 34 classes réellement aspectualisées. Le graphe nous indique que chaque fois que nous augmentons le seuil, le rappel augmente. Cela veut dire que chaque fois que nous augmentons le seuil, le nombre des classes détectées non réellement aspectualisées diminue.

5.5.3.3 La technique Fan-in

5.5.3.3.1 Seuil

Le *Fan-in* d'une méthode *m* est défini comme le nombre de méthodes qui ont fait appel à la méthode *m*. Si nous fixons, par exemple, un seuil égal à 5, cela veut dire que nous considérons que les méthodes qui ont un *Fan-in* supérieur ou égal à 5 comme étant des aspects candidats. Pour appliquer cette technique sur *JHotDraw*, nous avons utilisé *Fint*, un *plug-in* sous *Eclipse*. Voici ce que nous avons obtenu comme résultats :

5.5.3.3.2 Résultats

Tableau 17 Résultats de la technique Fan-in (précision et rappel) en fonction du seuil.

Seuil	Classes détectées réellement aspectualisées	Classes détectées	Rappel	Précision
108	0	4	98,36	0,00
93	1	5	98,36	2,94
63	2	6	98,36	5,88
62	2	7	97,95	5,88
61	2	9	97,13	5,88
60	2	10	96,72	5,88
59	2	11	96,31	5,88
58	2	13	95,49	5,88
57	2	14	95,08	5,88
56	2	19	93,03	5,88
55	3	23	91,80	8,82
47	4	24	91,80	11,76
46	4	26	90,98	11,76
35	5	27	90,98	14,71
34	6	28	90,98	17,65
27	6	29	90,57	17,65
25	7	30	90,57	20,59
22	7	31	90,16	20,59
21	7	32	89,75	20,59
19	8	34	89,34	23,53
18	8	35	88,93	23,53
17	8	36	88,52	23,53
16	9	37	88,52	26,47
15	10	40	87,70	29,41
14	12	42	87,70	35,29
12	14	46	86,89	41,18
11	14	48	86,07	41,18
9	14	51	84,84	41,18
8	17	59	82,79	50,00
7	27	88	75,00	79,41
6	28	103	69,26	82,35
5	30	121	62,70	88,24

Voici la représentation graphique des précisions et des rappels trouvés en fonctions du seuil :

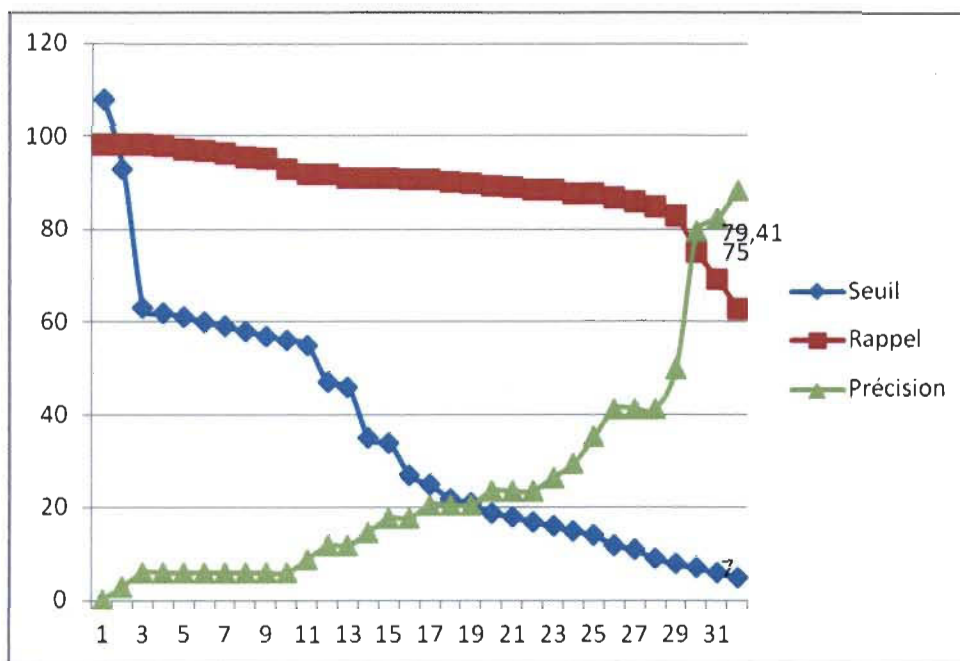


Figure 42 Précisions et rappels de la technique Fan-in en fonction du seuil.

5.5.3.3 Discussion

Nous remarquons dans le tableau ci-dessus que les précisions sont élevées au niveau des seuils 5, 6 et 7. La même chose pour le rappel. Il est entre 62% et 75%, ce qui signifie que l'outil n'a pas détecté un nombre important de méthodes non réellement aspectualisées. Le graphe nous montre le point optimal au niveau du seuil égal à 7. Nous avons une précision égale à 79,41% et un rappel égal à 75,00%.

5.5.3.4 Combinaison des résultats des techniques

5.5.3.4.1 Résultats

Tableau 18 Résultats des différentes combinaisons (précision et rappel).

techniques	Comparaison avec JHotDraw60b1			
	Nb classes	Nb classes asp.	Précision	Rappel
Tech1 \cap Tech2	23	9	26,47	94,26
Tech1 \cap Fan-in	52	13	38,24	84,02
Tech2 \cap Fan-in	31	16	46,06	93,85
Tech1 \cap Tech2 \cap Fan-in	14	8	23,53	97,54

5.5.3.4.2 Discussion

Nous observons que les rappels sont améliorés par rapport aux rappels des techniques appliquées séparément. Cela veut dire que le nombre de méthodes détectées non réellement aspectualisées a diminué, par contre les précisions sont trop faibles. La meilleure précision égale à 46,06 selon la combinaison $tech2 \cap Fan-in$ où l'intersection des résultats des deux techniques a donné 16 méthodes détectées et réellement aspectualisées parmi les 34 méthodes réellement aspectualisées.

5.5.4 Étude statistique

Nous avons utilisé l'outil *XSTAT* pour le calcul des corrélations de *Spearman* et de *Pearson* ainsi que l'AUC de la courbe ROC pour voir si les résultats des techniques ont une relation avec ce qui a été réellement aspectualisé ou non.

5.5.4.1 Résultats

La régression logistique est très utile lorsque nous voulons prédire l'effet d'une ou plusieurs variables sur une variable binaire (qui ne peut prendre que deux valeurs 0 ou 1). Pour notre cas, la variable réponse correspond à la colonne dans laquelle se trouve la variable binaire, autrement dit, la colonne contenant 278 cases (le nombre total des classes étudiées) où chaque case contient soit 0 (classe non réellement aspectualisée) ou 1 (classe réellement aspectualisée) et une autre variable dite explicative correspondant à la colonne dans laquelle se trouve le résultat binaire (1 pour les classes détectées et 0 pour les classes non détectées) de la technique ou la combinaison (intersection) des techniques étudiées. Nous avons résumé ce que nous avons obtenu dans le tableau suivant :

Tableau 19 Résultats des corrélations de Pearson et Spearman et l'AUC (courbe ROC).

techniques	Comparaison avec JHotDraw60b1		
	Pearson	Spearman	AUC (ROC)
Tech1	0,141	0,141	0,331
Tech2	0,340	0,340	0,460
Fan-in	0,337	0,337	0,553
Tech1 \cap Tech2	0,247	0,247	0,250
Tech1 \cap Fan-in	0,187	0,187	0,321
Tech2 \cap Fan-in	0,426	0,426	0,442
Tech1 \cap Tech2 \cap Fan-in	0,316	0,316	0,230

Selon les corrélations de *pearson* et de *spearman* des trois premières lignes (*Tech1*, *Tech2*, *Fan-in*), il existe une relation entre la variable explicative (classe détectée correspond à une case égale à 1, classe non détectée correspond à une case égale à 0) et la variable réponse (classe réellement aspectualisée correspond à une case égale à 1, classe

non réellement aspectualisée correspond à une case égale à 0) mais les valeurs de corrélations ne sont pas élevées. Pour la courbe ROC, l'*AUC* (aire sous la courbe) est entre 0.3 et 0.5, ce qui veut dire que le modèle est aléatoire.

Pour les résultats des différentes intersections, il existe aussi une relation entre ce que nous avons obtenu et ce qui a été aspectualisé malgré le fait que la relation n'est pas forte. Concernant la régression logistique, elle n'a pas donné un résultat intéressant à cause du fait que les modèles sont aléatoires.

La question qui peut attirer notre attention au niveau de la technique *Fan-in* est pourquoi nous avons obtenu une bonne précision (88,24 pour un seuil égal à 5), mais un modèle aléatoire avec une *AUC* égale à 0,553 ?

Réponse : Le nombre des classes détectées et non réellement aspectualisées (91 classes) a influencé négativement les corrélations et la courbe *roc*. Si nous supposons que la technique n'a détecté que 23 classes non réellement aspectualisées au lieu de 91 classes et nous recalculons les corrélations ainsi l'*AUC*.

Résultats :

Tableau 20 Corrélations et l'*AUC* de la technique *Fan-in* avec seulement 23 classes détectées non réellement aspectualisées.

techniques	Comparaison avec JHotDraw60b1		
	Pearson	Spearman	AUC (ROC)
Fan-in	0,701	0,701	0,869

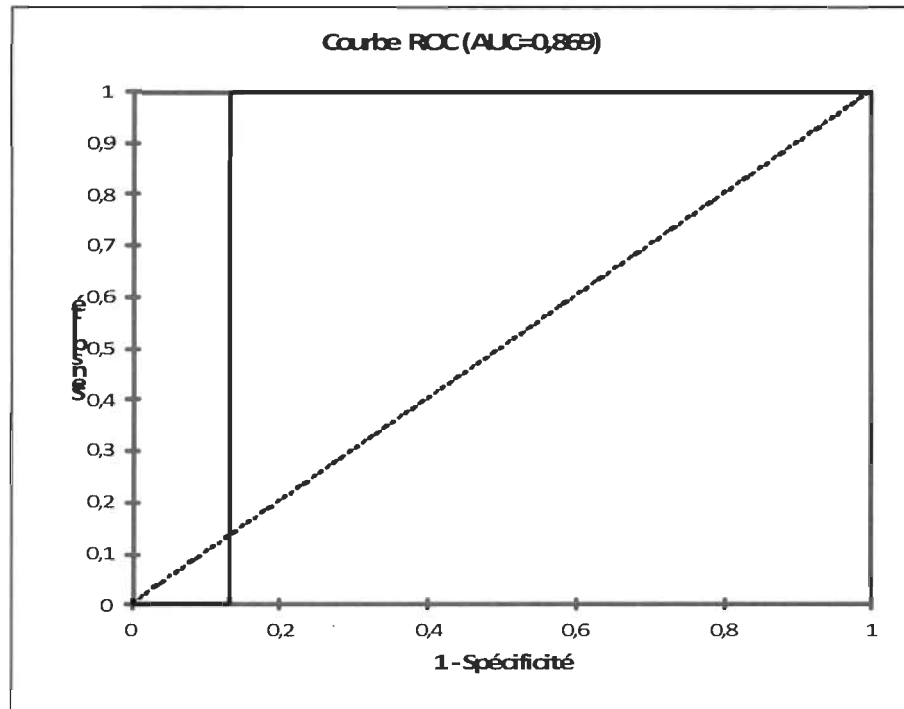


Figure 43 La courbe roc de la technique fan-in avec seulement 23 classes détectées et non réellement aspectualisées.

Les corrélations obtenues nous indiquent qu'il y a une relation positive et forte entre la variable réponse (classe réellement aspectualisée correspond à 1, classe non réellement aspectualisée correspond à 0) et la variable explicative (classe détectée par la technique *Fan-in* correspond à 1, classe non détectée par *Fan-in* correspond à 0), l'*AUC* nous montre que le modèle est excellent. Cela prouve que le nombre de classes détectées et non réellement aspectualisées a influencé négativement sur les corrélations et l'*AUC* montrées dans le tableau 3.

5.5.5 Conclusion

Les résultats des deux premières techniques étaient moyens, ce qui explique que l'aspectualisation de Jhotdraw n'était pas basée sur les appels récurrents ou sur la technique FCA avec ces deux contraintes. Par contre, nous avons pu avoir de bons résultats avec

l'application de la technique *Fan-in*, l'aspectualisation a donc touché un nombre important de classes contenant des méthodes dispersées. La même technique *Fan-in* a détecté un nombre de classes non réellement aspectualisées (91 classes). Cela a influencé négativement les corrélations ainsi que la courbe *roc*. Nous n'avons pas trouvé de bons résultats en comparaison avec ce qui a été aspectualisé réellement, cependant, les trois techniques et leurs résultats restent quand même intéressants. Si nous voulons diminuer le nombre des appels récurrents dans notre application, nous pourrions appliquer la première technique afin d'aspectualiser les appels récurrents et transverses détectés. Si nous voulons travailler avec la technique FCA ou avec la technique *Fan-in* pour détecter les méthodes dispersées cela reste aussi intéressant.

Chapitre 6 - Conclusion générale

Le but de ce travail était d'utiliser les principes de deux techniques dynamiques existantes, basées sur les traces d'exécutions (analyse des appels récurrents et analyse formelle de concepts), et les adapter à une analyse statique du code basée sur une analyse des chemins de contrôle. Le but principal de ce projet étant d'explorer une approche statique (par analyse du code) basée sur une analyse des chemins de contrôle pour la détection de code pouvant correspondre à des préoccupations transverses.

Nous avons observé à travers les trois exemples considérés et le benchmark JHotDraw que les deux nouvelles approches (adaptation des techniques considérées) ainsi que la technique *Fan-in* ont été en mesure d'identifier certaines préoccupations transverses réellement aspectualisées, mais que des différences significatives se posent pour d'autres préoccupations. Ces différences sont largement dues aux différentes façons dont les trois techniques s'appliquent.

Les résultats obtenus ne sont pas satisfaisants, mais cela ne veut pas dire que les techniques ont échoué dans la détection des éléments réellement aspectualisés, pour deux raisons importantes :

- ✓ Chaque technique a sa propre façon d'être appliquée et à quels types de préoccupations elle s'intéresse (appels récurrents, méthodes dispersée, etc.),

- ✓ Nous n'avons pas eu connaissance préalablement sur quels types de préoccupations transverses s'est basée l'aspectualisation de la version du programme JHotDraw étudié.

Cependant, à travers l'étude de cas présentée et à la lumière de l'évaluation faite sur les trois exemples utilisés dans le chapitre précédent, les résultats obtenus sont encourageants et représentent tout de même une validation des techniques adaptées.

Références

- [1] Gregor Kiczales, John Lamping, Anurag Mendhekar et Maeda, Cristina Lopes, Jean-Marc Longtier et Irwin, « Aspect-Oriented Programming », dans Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997), 1997.
- [2] Jean Baltus, La Programmation Orientée Aspect et AspectJ: Présentation et Application dans un Système Distribué. Facultés Universitaires, Notre-dame de la Paix, Namur, Belgique, jbaltus@info.fundp.ac.be
- [3] Bounour Nora, Ghouli Said and Atil Fadila, A Comparative Classification of Aspect Mining Approaches. Journal of Computer Science 2 (4) : 322-325, 2006.
- [4] Magiel Bruntink, Aspect Mining using Clone Class Metrics, Centrum voor Wiskunde en Informatica, The Netherlands, Magiel.Brunting@cw.nl
- [5] Andy Kellens, A Survey of Aspect Mining Tools and Techniques, Programming Technology Lab, Vrije Universiteit Brussel, Project IWT 040116 “AspectLab”, Workpackage 6 - Deliverable 6.2.a , June 30, 2005.
- [6] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, 2003.
- [7] K. Gybels and A. Kellens, Experiences with identifying aspects in smalltalk using “unique methods”. In Workshop on Linking Aspect Technology and Evolution, 2005.
- [8] M. Marin, A. Van Deursen, and L. Moonen, Identifying aspects using Fan-in analysis. In Working Conference on Reverse Engineering (WCRE), 2004.
- [9] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005), pages 13-22. IEEE Computer Society Press, 2005.
- [10] M. P. Robillard and G. C. Murphy, Concern graphs: Finding and describing concerns using structural program dependencies. In Proceedings of the 24th International Conference on Software engineering, pages 406-416. ACM Press, 2002.

- [11] K. Mens, B. Poll, and S. Gonzalez, Using intentional source-code views to aid software maintenance. In *Proceeding of the International Conference on Software Maintenance (ICSM'03)*, pages 169-178. IEEE Computer Society Press, 2003.
- [12] W. Griswold, Y. Kato, and J. Yuan, Aspect browser: Tool support for managing dispersed aspects. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems – OOPSLA 99*, 1999.
- [13] D. Janzen and K. De Volder, Navigating and querying code without getting lost. In *International Conference on Aspect Oriented Software Development 2003*, 2003.
- [14] S. Breu and J. Krinke, Aspect mining using event traces. *Int. Conference on Automated Software Engineering (ASE)*, Septembre 2004.
- [15] P. Tonella and M. Ceccato, Aspect mining through the formal concept analysis of execution traces. In *11th IEEE Working Conference on Reverse Engineering*, 2004.
- [16] D. Shepherd, T. Tourwé, and L. Pollock, Using langage clues to discover crosscutting concerns. In *Workshop on the Modeling and Analysis of Concerns*, 2005.
- [17] Daniel St-Yves, Dépendences et Gestion des Modifications dans les Systèmes Orientés Objet, Mémoire présenté à l'université du Québec à Trois-Rivières, décembre 2007.
- [18] Tom Tourwé et Kim Mens, Mining Aspectual Views using Formal Cncept Analysis. In *Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*.
- [19] Priss Uta, Formal Concept Analysis in Information Science. Napier University, USA, This is a draft version of a paper to be pulished in Cronin, Blaise (Ed.), *Annual Review of Information Science and Technology*, ASIST, Vol. 40. 2006, vol. 40, pp. 521-543.
- [20] Àlvaro Garcia Pérez, A Functional Approach to the Observer Pattern, Oxford, May 2009.
- [21] Magiel Bruntink, Aspect Mining using Clone Class Metrics. Centrum voor Wiskunde en Informatica, The Netherlands.
- [22] Marius Marin, Leon Moonen and Arie van Deursen, A common framework for aspect mining based on crosscutting concern sorts. Delft University of Technology, Software Engineering Research Group Technical Report Series. Report TUD-SERG-2006-009.
- [23] JHotDraw Home Page. www.jhotdraw.org.

- [24] Arie van Deursen, Marius Marin and Leon Moonen, AJHotDraw: A showcase for refactoring to aspects (2005).
- [25] John Makhoul, Francis Kubala, Richard Schwartz, Ralph Weischedel. Performance Measures for Information Extraction, in Proceedings of DARPA Broadcast News Workshop, Herndon, VA, February 1999.
- [26] Ricco Rakotomalala, Analyse de corrélation, Étude des dépendances – Variables quantitatives. Université Lumière Lyon 2 (2012).
- [27] <http://www.jybaudot.fr>
- [28] H. Delacour, A. Servonnet, A. Perrot, J.F. Vigezzi and J.M Ramirez, La courbe ROC (receiver operating characteristic) : principes et principales applications en biologie clinique. Ann Biol Clin 2005.

Annexe A – Code source de l'exemple 1 (Design Pattern Observer) du chapitre 5

```
//la classe GuiElement
import java.awt.Color;
import java.util.ArrayList;
import java.util.Iterator;

public class GuiElement {
    private Color color;
    private ArrayList observers = new ArrayList();
    public void setColor(Color color){
        this.color = color;
        this.informObservers();
    }

    public void attachObservers(Observer observer){
        observers.add(observer);
    }

    public void detachObservers(Observer observer){
        observers.remove(observer);
    }

    public void informObservers(){
        for (Iterator
            it=observers.iterator();it.hasNext();)
            ((Observer) it.next()).update();
    }
}
```

```

//la classe Point
import java.util.ArrayList;
import java.util.Iterator;

public class Point {
    private int x;
    private int y;
    private ArrayList observers = new ArrayList();
    public void setX(int x){
        this.x = x;
        this.informObservers();
    }

    public void setY(int y){
        this.y = y;
        this.informObservers();
    }

    public void setXY(int x, int y){
        this.x = x;
        this.y = y;
        this.informObservers();
    }

    public void attachObservers(Observer observer){
        observers.add(observer);
    }

    public void detachObservers(Observer observer){
        observers.remove(observer);
    }

    public void informObservers(){
        for (Iterator
            it=observers.iterator();it.hasNext();)
            ((Observer) it.next()).update();
    }
}

```

```
//la classe Main
public class Main {

    public static void main(String[] args) {
        Point p1 = new Point ();
        p1.setX(4);
        p1.setY(3);
        Point p2 = new Point();
        p2.setXY(5,6);
        p2.setY(7);
    }
}
```

Annexe B – Code source de l'exemple 2 (la classe TangledStack) du chapitre 5

```
//la classe Main
public class Main {
    public static void main(String args[]){
        JFrame f = new JFrame();
        Tangledstack tangledstack = new
        Tangledstack(f);
        Object tab[] = {1,2,3,4,5,6,7,8,9,10};
        tangledstack._elements.equals(tab);
        tangledstack.push
        (tangledstack._elements[0]);
        tangledstack.pop();
        tangledstack.top();
    }
}
```

```

//la classe Tangledstack
import javax.swing.*;
public class Tangledstack {
    private int _top = -1;
    private Object[] _elements;
    private final int S_SIZE = 10;
    private JLabel _Label = new JLabel("stack");
    private JTextField _text = new JTextField(20);
    public Tangledstack(JFrame frame) {
        _elements = new Object[S_SIZE];
        frame.getContentPane().add(_Label);
        _text.setText("[]");
        frame.getContentPane().add(_text);
    }
    public String toString() {
        StringBuffer result = new StringBuffer("");
        for(int i = 0; i <= _top; i++){
            result.append(_elements[i].toString());
            if(i != _top)
                result.append(", ");
        }
        result.append("]");
        return result.toString();
    }
    private void display() {
        _text.setText(toString());
    }
    public void push(Object element) {
        if (!isFull()){
            _elements[++_top] = element;
            display();
        }
        else
            System.out.println("la pile est pleine");
    }
    public void pop() {
        if (!isEmpty()){
            _top--;
            display();
        }
        else
            System.out.println("la pile est vide");
    }
    public Object top() {
        if (isEmpty())
            System.out.println("la pile est vide");
        return _elements[_top];
    }
    public boolean isFull(){
        return (_top == S_SIZE - 1);
    }
    public boolean isEmpty(){
        return (_top < 0);
    }
}

```

Annexe C – Code source de l'exemple 3 (Design pattern chaîne de responsabilité) du chapitre 5

```

import java.awt.Color;
import javax.swing.*;
import javax.swing.border.*;
public class ColorImage extends JPanel implements
Chain{
    private Chain _nextChain;
    public ColorImage() {
        super();
        setBorder(new LineBorder(Color.black));
    }
    public void addChain(Chain c) {
        _nextChain = c;
    }
    public void sendToChain(String msg){
        Color c = getColor(msg) ;
        if (c != null) {
            setBackground(c);
        }
        else {
            if(_nextChain !=null)
                _nextChain.sendToChain(msg);
        }
    }
    private Color getColor(String msg) {
        String lmesg = msg.toLowerCase();
        Color c = null;
        if(lmesg.equals("red"))
            c = Color.red;
        if(lmesg.equals("blue"))
            c = Color.blue;
        if(lmesg.equals("green"))
            c = Color.green;
        return c;
    }
    public Chain getChain(){
        return _nextChain ;
    }
}

```

```
public interface Chain {  
    public abstract void addChain(Chain c);  
    public abstract void sendToChain(String msg);  
    public Chain getChain();  
}
```